Broadview
www.broadview.com.cn

MANNING

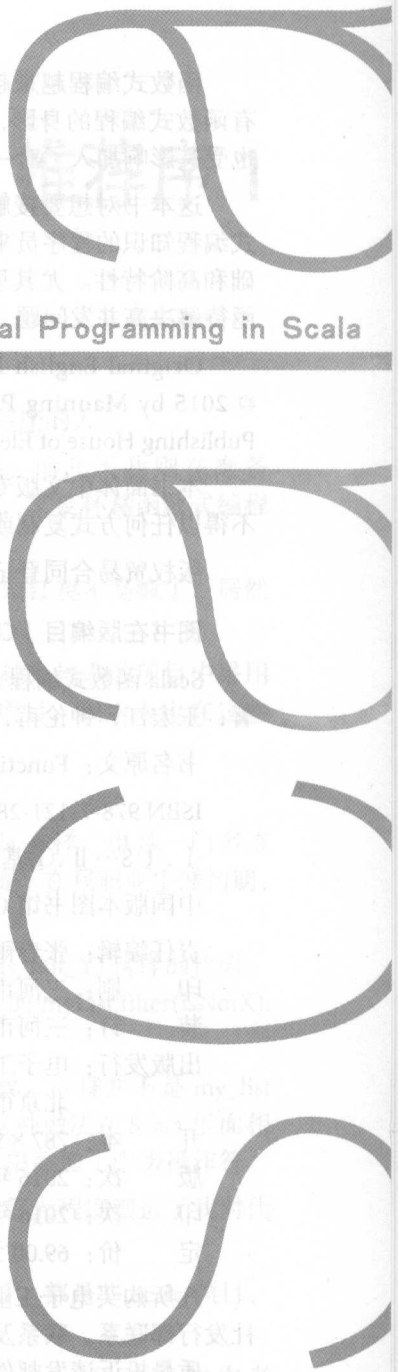
Functional Programming in Scala

Scala 之父 Martin Odersky 作序

Scala函数式编程

[美] Paul Chiusano　Rúnar Bjarnason　著
王宏江　钟伦甫　曹静静　译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn



Functional Programming in Scala

Scala函数式编程

[美] Paul Chiusano　Rúnar Bjarnason　著

王宏江　钟伦甫　曹静静　译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

函数式编程越来越多地从学术界走向工业界，很多和人们日常相关的重要系统背后都有函数式编程的身影，并且比例越来越高。在这种大的趋势下，甚至很多指令式编程语言也受其影响加入了对一些函数式特征的支持，比如 Java 8 终于将 lambda 加了进来。

这本书对想要接触函数式编程，或在实际业务中已经使用函数式但想要系统巩固函数式编程知识的程序员来说，是一本非常有价值的书。它以 Scala 为载体，涵盖了函数式的基础和高阶特性。尤其里面的一些高阶特性是在其他书籍中极少介绍到的。因而，非常值得亟待解决高并发问题，或大数据领域从业的开发人员学习。

版权贸易合同登记号 图字：01-2015-3996

图书在版编目（CIP）数据

Scala 函数式编程 /（美）基乌萨诺（Chiusano, P.），（美）比亚尔纳松（Bjarnason, R.）著；王宏江，钟伦甫，曹静静译 . — 北京：电子工业出版社，2016.4

书名原文：Functional Programming in Scala

ISBN 978-7-121-28330-7

Ⅰ .①S… Ⅱ .①基…②比…③王…④钟…⑤曹… Ⅲ .①JAVA 语言 – 程序设计Ⅳ .①TP312

中国版本图书馆 CIP 数据核字 (2016) 第 053663 号

责任编辑：张春雨
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787 × 980　1/16　印张：16.75　　字数：402 千字
版　　次：2016 年 4 月第 1 版
印　　次：2016 年 4 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

推荐序 1

我可能是全中国程序员圈子里最不适合给《Scala 函数式编程》写序的人。

三年前我写过《Scala 七大死穴》，算是把 Scala 批判了一番。前几天我则在准备 ThoughtWorks 咨询师大会上的讨论话题《没有函数的函数式编程》，又要杯葛函数式编程的样子。

看起来，我无论对 Scala 还是对函数式编程，都没什么好评嘛。宏江莫不是疯了，居然要我来写序？

等等，事情似乎不是这样。最近几年，ThoughtWorks 的客户在越来越多的项目中采用了 Scala 技术栈，ThoughtWorks 也孵化出了若干基于 Scalaz 的开源项目。我本人也在这些项目中起到了一些作用。

为什么我会做这些“口嫌体正直”的事呢？这得从十年前说起。

我最早是在 C++ 中开始接触到函数式编程的概念。C++ 和 Scala 一样，也是一门多范式语言。C++ 的标准库和 Boost 都提供了许多函数式编程的设施。但是，在我职业生涯初期，给我留下深刻印象的函数式编程库要数 Boost.Egg。

利用 Boost.Egg，你可以写出 my_list|filtered(&is_not_X)|filtered(&is_not_Y) 这样的代码。你会注意到这种用法和 Scala 标准库非常相像，它大致相当于 Scala 的 myList.filter(isNotX).filter(isNotY)，这种 filter 的用法，本书第 5 章中也有讲解。

Boost.Egg 的另一个特点是“非侵入”，比如上例的 filtered 函数，本身并不是 my_list 的成员。相反，我们通过重载 | 运算符给原有的类型添加新功能。这种做法在 Scala 里面相当于隐式转换，本书第 9 章中提供的例子正是利用隐式转换，给字符串添加了中缀操作符。

虽然 Boost.Egg 没能流行起来，但对我个人而言很重要，因为它很大程度塑造了我对代码的品位。

有趣的是，Boost.Egg 的作者 Shunsuke Sogame 近年来的开源项目，都是些 Scala 项目，可能这也是因为 C++ 和 Scala 非常相似的缘故吧。

另一个对我代码品位影响很大的技术是 Lua 中的协程（coroutine）。Lua 的作者 Roberto Ierusalimschy 把协程称为“单趟延续执行流”（One-shot continuation）。有了协程

或者延续执行流，程序员可以手动切换执行流，不再需要编写事件回调函数，而可以编写直接命令式风格代码但不阻塞真正的线程。我的前东家网易在开发游戏时，会大量使用协程来处理业务逻辑，一个游戏程序内同一时刻会运行成千上万个协程。

而在其他不支持协程或者延续执行流的语言中，程序员需要非阻塞或异步编程时，就必须采用层层嵌套回调函数的 CPS（Continuation-Passing Style）风格。这种风格在逻辑复杂时，会陷入“回调地狱”（Callback Hell）的陷阱，使得代码很难读懂，维护起来很困难。

Scala 语言本身并不支持协程或者延续执行流。因此，一般来说，程序员需要非阻塞或异步编程时，就必须使用类似本书第 13 章“外部作用和 I/O”中介绍的技术，注册回调函数或者用 for/yield 语句来执行异步操作。如果流程很复杂的话，即使是 for/yield 语法仍然会陷入回调地狱。

我对 Scala 开源社区的贡献之一是 stateless-future。这个库提供了一些宏，实现了延续执行流，可以自动把命令式风格的代码转换成 CPS 风格。通过这种做法，程序员不再需要手写本书 13.2 节那样的代码了，编写的代码风格更像普通的 Java 或者 PHP 风格，直接像流水账一样描述顺序流程。

后来，我把这种避免回调函数的思路，推广到了其他用途上。比如，我开发了基于 Scala.js 的前端框架 Binding.scala。使用 Binding.scala 的用户，编写普通的 HTML 模板，描述视图中的变量绑定关系，而不需要编写高阶函数就能做出交互复杂的网页。

而我的另一个开源库 Each，则更进一步，支持一切 monad。大多数情况下，使用了 Each 就不需要编写任何高阶函数，我称之为“没有函数的函数式编程”。这意味，本书第 11 章到第 15 章的全部内容，你都可以直接编写类似 Java 的命令式语法，而 Each 则自动帮你生成使用 monad 的代码。

总之，我是 Scala 函数式编程的死对头，我写的 Scala 库，恰恰是为了避免使用本书中谆谆教导的各种高阶函数。如果你是个 Java 程序员，想在最短的时间内用 Scala 开始“搬砖”，那么，从实用角度出发，我建议你合上本书，直接用 Each 即可。因为，虽然 Each 最终会生成 Monad 风格代码，但是，本书中涉及的使用高阶函数的细节，就像汇编语言一样，就算你不知道也照样可以干活。

不过，如果你是个求道者，追求编程艺术的真理，希望刨根到底，理解函数式编程的内在理论和实现机制，那么本书很适合你。

这本书绝不轻易放过每个知识点，全书包含有大量习题，要求你自己实现 Scala 标准库或者 Scalaz 中的既有功能。所以，当你读完本书，做完习题后，虽然你的应用开发能力并不会直接提升，但你会体会到构建函数式语言和框架时的难点和取舍，从而增进你的框架开发和语言设计的能力。

ThoughtWorks Lead Consultant　杨博

参考资料

1. Boost.Egg
2. 关于 Lua 中的协程，参见 A. L. de Moura, N. Rodriguez, and R. Ierusalimschy. Coroutines in Lua. Journal of Universal Computer Science, 10(7):910–925.
3. 关于延续执行体的历史，参见 Reynolds, John C. (1993). “The discoveries of continuations” (PDF). Lisp and Symbolic Computation 6 (3/4): 233–248.
4. 关于 Scala 异步编程的“回调地狱”问题，参见 Business-Friendly Functional Programming – Part 1: Asynchronous Operations

推荐序 2

函数式编程与命令式编程同样源远流长，然而在计算机应用的历史进程中，二者的地位却颇不对等。命令式编程几乎自始至终都是大众的宠儿，函数式编程却长期局限于象牙塔和少数应用领域之内。尽管如此，函数式编程的重要性却从未被忽视，几十年来生机勃勃地发展，静静地等待着逆袭的时刻。事实上，即便是浸淫于命令式编程多年的工程师，也常常会与函数式编程亲密接触而不自知：例如 SQL、C++ 模板元编程，还有 C++ 标准库中的 STL 等，多少都带有一些函数式的色彩。早年，受软硬件水平的限制，函数式语言缺乏高效的编译器和运行时支持，这可能是函数式语言错失流行机会的一大原因。近年来，一方面程序语言理论和实现技术突飞猛进，函数式语言在性能上的劣势越来越不明显；另一方面，随着多核、高并发、分布式场景激增，大众也逐渐开始认识到函数式编程在这些领域得天独厚的优势。然而，流连于主流命令式语言多年积攒下的库、框架、社区等丰富资产，再加上长期的教育惯性与思维惯性，使得人们仍然难以在生产上完全转向函数式语言。

一个新的契机，来自于大数据。社交网络、个人移动设备、物联网等新技术的兴起，使得海量数据处理开始成为家常便饭。人们突然发现，自己在命令式世界的武器库中，竟找不出称手的兵器来攻打大数据这座堡垒。2008 年，Google 发表了跨时代的 MapReduce 论文。尽管学界对 MapReduce 颇有非议[1]，MapReduce 的核心思想仍然旋风般席卷了整个工业界，为大数据技术的发展带来了及时而深远的影响。有趣的是，MapReduce 的核心思想，正是来自于天生擅长高并发和分布式场景的函数式编程。自此以后，各色大数据处理系统层出不穷，而其中的函数式成分也愈加浓重：在用户接口层面，这些系统往往以 DSL 的形式提供一套类 SQL 语义、具函数式特征的申明式查询接口；在实现层面，则仰仗不变性等约束来简化并发和容错。然而，出于种种原因，大部分系统的实现语言仍然以 C++、Java、C# 这些命令式语言为主。可谓操命令式之形而施函数式之实。

自 2009 年起，我先后接触了 Erlang、Scheme、ML 等函数式语言。但出于显而易见的原因，未能有机会将之用于工程实战。2013 年春节前后，我参加了由 Scala 之父 Martin Odersky 在 Coursera 上开设的 Functional Programming Principles in Scala 课程。凑巧的是，就在课程结束后不久，我便得到一个机会加入 Intel 参与有关大数据和 Apache Spark 的工作。函数式语

1 David J. DeWitt and Michael Stonebraker, MapReduce: A major step backwards, https://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html

言和分布式系统一直是我的两大兴趣点，由 Scala 开发的 Spark 恰恰是二者的一个完美融合。于是，Scala 便成了我的第一门实战函数式语言。近年来 Spark 的火爆，更是对 Scala 和函数式编程的推广起到了推波助澜的作用。

与我所熟悉的其他函数式语言相比，我想 Scala 最大的优点之一就是过渡平滑。立足于 JVM 并将函数式融入如日中天的面向对象，这样的设计带来了两大明显的好处。第一，顺畅地集成了 Java 社区多年积累的财富。第二，Scala 和 C++ 类似，也是一门“广谱”多范式语言；不熟悉函数式编程的 Scala 初学者，随时可以安全地回退，转用熟悉的命令式面向对象范式编程，从而保证交付速度。这个设计的背后，应该与 Martin Odersky“学术工业两手抓、两手都很硬”的风格不无关系。论学术，他师承 Pascal 之父 Niklaus Wirth，在代码分析和程序语言设计方面建树颇丰；论工业应用，他一手打造了 Generic Java 和新一代的 `javac` 编译器。可以说 Martin 既具备了用以高瞻远瞩的理论基础，又十分了解普罗大众的痛点。两相结合，这才造就了 Scala 这样一个平衡于阳春白雪和下里巴人之间的作品。

其实函数式编程本身并没有多难。对于接受过若干年数学训练，却没有任何编程经验的人来说，相较于命令式编程中的破坏性赋值，函数式编程中的不变性、递归等概念反而应该更加亲切。譬如中学做证明题时，我们从不会先令 a = x，隔上几行再令 a = a + 1。真正的困难在于，函数式编程中的一些概念和手法——如用尾递归代替循环——与命令式编程的直觉相冲突。对于那些有着多年命令式语言编程经验，把 Java 用得比母语还溜的工程师们而言，一边要学习新的知识，一边还要克服多年编程训练所造成的思维定势，无异于逆水行舟。而在 Scala 中，你永远有机会在实战中回退至自己的舒适区。实际上，我们完全可以无视 Scala 的函数式特性，仅仅将 Scala 当作语法更加洗练的 Java。因此，对于那些操持主流命令式面向对象语言多年的工程师们而言，Scala 特别适合用作涉猎函数式编程的起步语言。

这本书所讲授的，正是基于 Scala 的函数式编程基础。基于 Scheme、Haskell 等老牌函数式语言的传统教材的问题在于，相关语言的语法和思维方式与读者现有的知识体系迥异，容易造成较为陡峭的入门门槛。此外，由于这些语言本身的实际应用机会不多，初学者也难以在实战中获得宝贵的直觉和经验。而在 Scala 的帮助下，这本书并不要求你抛开现有的思维方式另起炉灶，它所做的更像是为你现有的思维方式添砖加瓦，从而令你如虎添翼。

最后，作为曾经的译者，我深知在国内当前的大环境下技术翻译之不易。每一本优秀技术书籍的中译本背后都是译者数月乃至十数月的艰苦付出。感谢诸位译者为我们带来又一本好书！

Spark committer from Databricks 连城

译序

在近些年新兴起的一些编程语言里，带有函数式特性的语言占据很大的比例，其中Scala的发展无疑是非常突出的。目前来看Scala所取得的成功，主要归功于它抓住了互联网的快速发展、大数据的兴起，以及部分软件企业的产业升级等机遇（尤其在大数据领域，Scala已经成为这个领域的明星语言）；这些领域的发展变化对语言提出了新的要求：提升生产力，满足高并发、高性能及高度抽象——Scala正好都可以满足。Scala内置了很多设计模式（以语法糖的方式），也吸取了很多其他编程语言里已经被验证的优点，比如类型推导、模式匹配等，这极大地简化了开发过程中所要写的代码量（相对于Java），提升了程序员的生产效率。此外，得益于JVM本身的强大威力，加上编译器的一些优化，使它在运行时的效率与Java并没有太大的差距。

作为深受学院派风格影响的语言，Scala本身并不像很多经典的函数式语言那么“纯粹”，而是很大程度地迎合了工业界的需求——保证了与Java极大的兼容。程序员既可以用传统的Java风格（指令式）实现功能，也可以用函数式风格来实现，这为更多程序员提供了相对平滑的过渡方式。

但是即便Scala提供了面向Java程序员的相对平滑的适应方式，依然让很多程序员觉得它有很多高深复杂的地方，即使一个有多年Scala编程经验的程序员也可能对它的很多特性或用法不甚了解。一方面是Scala吸纳百家所长，特性有些多，另一方面是大家对函数式编程了解得还不够。函数式编程的历史很久远，但一直只在业界某些特定领域使用，并未被广泛接受，而最近十余年它才开始逐渐升温。在这之前不管是大学里的普通计算机教育，还是工作实践，多是以C++/Java等指令式编程为主的。大多数程序员并没有对函数式进行系统的了解。函数式编程是一个范围宏大的话题，它像一个参差多态的森林，很多人进入其中也仅是只见树木，不见森林。

五六年前接触Scala的时候，刚开始只是把它当作改良的Java，在一些非核心应用上做了一些简单尝试，并未有特别的感受。直到某天遇到了monad，这个概念一下把我难住了。费了很大的精力才逐渐明白它是怎么回事，之后反思为何这个问题难以理解，很大原因是以往的经验困扰了自己，总是将新概念与原有的经验去匹配，而它事实上并不能与你既有的世界观相匹配。原来还可以用这种思维去编程！于是开始清空以往固有的认识，严肃地去了解函数式和类型系统。在学习类型系统的过程中，发现了RÚNAR的博客，他在博客

上介绍了很多函数式编程的技巧和模式。这些模式不同于我们熟悉的面向对象里常见的设计模式，它多是从类型系统的角度去抽象的，这给我们带来很多新的思考。后来 RÚNAR 把这些内容总结成了这本书，也是机缘巧合，在去年通过老高跟引入这本书中文版权的电子工业出版社取得联系，最终我和同事钟伦甫、曹静静共同完成了对这本书的翻译。

为什么这些大数据的开发者会用这门语言制造了 spark、kafka、algebird/summingbird 等流行产品？并不是说其他语言不可以，而是这些开发者大多都是有学院背景且非常聪明的一群人，他们选择 Scala 除了品味之外，还基于它的抽象能力和它的表达能力。Scala 在这方面比 Java 等语言具备更高的抽象能力，很大一部分原因是它继承自 Haskell 和 ML 里的一些概念。发明 Haskell 的这群数学家为我们提供了另一种抽象问题的方式，这里面有众多的概念都在编程语言领域影响深远。

尽管函数式编程在近十多年用得越来越多，但市面上介绍其高阶特性的书却并不多。这本书在这方面是个重要的补充，它不仅仅面向 Scala 程序员，同样面向用任何编程语言开发的程序员，只要你充满好奇心。

真正参与一本书的翻译之后才发现翻译是要求很高的事，并不只是逻辑上正确与否的问题。这个过程非常感谢高宇翔（程序员老高），以及电子工业出版社的编辑在翻译过程中给出的很多帮助和指正。Scala 领域还有一些英文术语没有形成固定的中文叫法，我们尽量参考了市面上其他几本 Scala 书里的一些叫法。因为水平有限，难免会有错误，恳请读者在发现任何存疑的地方时给予反馈，可以通过邮箱（w.hongjiang@gmail.com）或微博、博客等方式联系我。

王宏江

目录

原推荐序 XVI
序言 XVIII
致谢 XIX
关于本书 XX

第一部分　函数式编程介绍　1

1　什么是函数式编程 2

1.1　函数式编程的好处：一个简单的例子 3
1.1.1　一段带有副作用的程序 3
1.1.2　函数式的解法：去除副作用 5
1.2　（纯）函数究竟是什么 7
1.3　引用透明、纯粹度以及替代模型 8
1.4　小结 10

2　在 Scala 中使用函数式编程 11

2.1　Scala 语言介绍：一个例子 11
2.2　运行程序 14
2.3　模块、对象和命名空间 15
2.4　高阶函数：把函数传给函数 15
2.4.1　迂回做法：使用循环方式 16
2.4.2　第一个高阶函数 17
2.5　多态函数：基于类型的抽象 18
2.5.1　一个多态函数的例子 19
2.5.2　对高阶函数传入匿名函数 20
2.6　通过类型来实现多态 21
2.7　小结 23

3　函数式数据结构 24

3.1　定义函数式数据结构 24

3.2 模式匹配.....26
3.3 函数式数据结构中的数据共享.....29
3.3.1 数据共享的效率.....30
3.3.2 改进高阶函数的类型推导.....31
3.4 基于 list 的递归并泛化为高阶函数.....32
3.4.1 更多与列表相关的函数.....35
3.4.2 用简单组件组合 list 函数时的效率损失.....36
3.5 树.....37
3.6 小结.....39

4 不是用异常来处理错误.....40

4.1 异常的优点与劣势.....40
4.2 异常的其他选择.....42
4.3 Option 数据类型.....43
4.3.1 Option 的使用模式.....44
4.3.2 Option 的组合、提升及对面向异常的 API 的包装.....47
4.4 Either 数据类型.....50
4.5 小结.....52

5 严格求值和惰性求值.....54

5.1 严格和非严格函数.....55
5.2 一个扩展例子：惰性列表.....57
5.2.1 对 Stream 保持记忆，避免重复运算.....58
5.2.2 用于检测 Stream 的 helper 函数.....59
5.3 把函数的描述与求值分离.....59
5.4 无限流与共递归.....62
5.5 小结.....65

6 纯函数式状态.....66

6.1 以副作用方式生成随机数.....66
6.2 纯函数式随机数生成器.....67
6.3 用纯函数式实现带状态的 API.....69
6.4 状态行为的更好的 API.....71
6.4.1 组合状态行为.....71
6.4.2 嵌套状态行为.....72
6.5 更通用的状态行为数据类型.....74
6.6 纯函数式命令编程.....74
6.7 小结.....76

第二部分 功能设计和组合子库 77

7 纯函数式的并行计算 78

7.1 选择数据类型和函数 79
7.1.1 一种用于并行计算的数据类型 80
7.1.2 组合并行计算 82
7.1.3 显性分流 83
7.2 确定表现形式 85
7.3 完善 API 86
7.4 API 与代数 89
7.4.1 映射法则 90
7.4.2 分流法则 91
7.4.3 打破法则：一个微妙的 bug 92
7.4.4 用 Actor 实现一个完全无阻塞的 Par 94
7.5 完善组合子为更通用的形式 98
7.6 小结 100

8 基于性质的测试 101

8.1 基于性质测试概览 101
8.2 选择数据类型和函数 103
8.2.1 API 的初始代码片段 103
8.2.2 性质的含义与 API 104
8.2.3 生成器的意义和 API 106
8.2.4 生成值决定生成器 107
8.2.5 精炼 Prop 的数据类型 108
8.3 最小化测试用例 109
8.4 使用库并改进其易用性 111
8.4.1 一些简单的例子 111
8.4.2 为并行计算编写测试套件 112
8.5 测试高阶函数及展望未来 116
8.6 生成器法则 117
8.7 小结 118

9 语法分析器组合子 119

9.1 代数设计，走起 120
9.2 一种可能的代数 124
9.2.1 切片和非空重复 125
9.3 处理上下文的相关性 127

9.4 写一个 JSON 分析器 128
9.4.1 JSON 格式 129
9.4.2 JSON 分析器 130
9.5 错误提示 130
9.5.1 一种可行的设计 131
9.5.2 错误嵌套 132
9.5.3 控制分支和回溯轨迹 133
9.6 实现代数 134
9.6.1 一种可能的实现 135
9.6.2 串化分析器 136
9.6.3 标记分析器 137
9.6.4 故障转移和回溯 138
9.6.5 上下文相关的分析 138
9.7 小结 140

第三部分 函数设计的通用结构 141

10 Monoid 142
10.1 什么是 monoid 142
10.2 使用 monoid 折叠列表 144
10.3 结合律和并行化 145
10.4 例子：并行解析 147
10.5 可折叠数据结构 148
10.6 组合 monoid 149
10.6.1 组装更加复杂的 monoid 150
10.6.2 使用组合的 monoid 融合多个遍历 151
10.7 小结 151

11 Monad 152
11.1 函子：对 map 函数的泛化 152
11.1.1 函子法则 153
11.2 Monad：对 flatMap 和 unit 函数的泛化 154
11.3 Monadic 组合子 157
11.4 单子定律 158
11.4.1 结合法则 158
11.4.2 为指定的 monad 证明结合法则 159
11.4.3 单位元法则 160

11.5 什么是 monad 161
11.5.1 identity monad 162
11.5.2 状态 monad 和 partial type application 163
11.6 小结 166

12 可应用和可遍历函子 167

12.1 泛化单子 167
12.2 Applicative trait 168
12.3 单子与可应用函子的区别 170
12.3.1 对比 Option applicative 与 Option monad 170
12.3.2 对比 Parser applicative 与 Parser monad 171
12.4 可应用函子的优势 172
12.4.1 不是所有的可应用函子都是 Monad 173
12.5 可应用法则 175
12.5.1 Left and right identity 175
12.5.2 结合律 176
12.5.3 Naturality of product 176
12.6 可遍历函子 178
12.7 使用 Traverse 179
12.7.1 从 monoid 到可应用函子 180
12.7.2 带状态的遍历 181
12.7.3 组合可遍历结构 182
12.7.4 遍历融合 183
12.7.5 嵌套遍历 184
12.7.6 Monad 组合 184
12.8 小结 185

第四部分 作用与 I/O 187

13 外部作用和 I/O 188

13.1 分解作用 188
13.2 一个简单的 IO 类型 189
13.2.1 处理输入效果 190
13.2.2 简单 IO 类型的优缺点 194
13.3 避免栈溢出 194
13.3.1 将一个控制流转化为数据构造子 195
13.3.2 Trampolining：栈溢出的通用解决方法 197

13.4 一个更微妙的 IO 类型 198
13.4.1 合理的 monad 199
13.4.2 一个支持控制台 I/O 的 monad 200
13.4.3 纯解释器 202
13.5 非阻塞和异步 I/O 204
13.6 一个通用的 IO 类型 206
13.6.1 最终的 main 程序 206
13.7 为什么 IO 类型不足以支撑流式 I/O 207
13.8 小结 209

14 本地影响和可变状态 210

14.1 纯函数式的可变状态 210
14.2 一种限制副作用范围的数据类型 212
14.2.1 受限可变性的语言表达 212
14.2.2 一种可变引用的代数表达 214
14.2.3 执行修改状态的行为 215
14.2.4 可变数组 217
14.2.5 一个纯函数的 in-place 快排实现 218
14.3 纯粹是相对于上下文的 219
14.3.1 副作用是什么？ 221
14.4 小结 222

15 流式处理与增量 I/O 223

15.1 命令式 I/O 的问题示例 223
15.2 一个简单的流转换器 225
15.2.1 创建 Process 227
15.2.2 组合和追加处理 229
15.2.3 处理文件 231
15.3 可扩展的处理类型 232
15.3.1 来源 234
15.3.2 保证资源安全 235
15.3.3 单一输入过程 237
15.3.4 多个输入流 239
15.3.5 去向 241
15.3.6 Effectful 通道 242
15.3.7 动态资源分配 243
15.4 应用场景 244
15.5 小结 245

原推荐序

函数式编程作为书题出现在 Scala 中是个有趣的现象。毕竟，通常 Scala 被称为函数式编程语言，而且在市场上有非常多的 Scala 相关书籍。是不是这些书都缺失了对语言函数式方面内容的描述？为了回答这个问题，我们需要有指导性地进行深挖。什么是函数式编程？对我来说，它是“使用函数编程”的别名，换句话说，是一种聚焦在函数上的编程方式。那么什么是函数？再来探寻更大范围的定义。当一种定义承认函数可能有副作用并返回结果时，纯函数式编程限制函数就像数学里定义的那样：用一种二元关系去映射参数到结果。

Scala 是不纯粹的函数式编程语言，它同时承认非纯粹函数和纯函数，而且没有使用不同的语法或给予不同的类型去区分这两种函数种类。其他函数式语言也有同样的属性。在 Scala 里如果能区分纯函数和非纯函数将是很好的，但我认为我们没有找到轻量级的和无需迟疑的灵活方式来这么做。

可以确信的是，Scala 程序员是被鼓励使用纯函数编程的。副作用也有，比如易变、I/O 或者异常的使用没有被禁止，事实上这些副作用有的时候使用起来十分方便，使用它们的原因有互用性、高效、方便等。但是专家的建议是过度地使用副作用普遍来说不是一种好的方式。然而，因为在 Scala 中非纯函数编程是可能的甚至是方便的，对命令式编程背景的程序员来说，保持他们的风格和不努力采用函数式思维的诱惑就非常大了。事实是，非常有可能将 Scala 编写成没有封号结尾的 Java 程序。

那么要学习 Scala 中的函数式编程，是不是需要先学习纯函数式编程，比如 Haskell？任何关于方法的争论都在本书的出现后被极大地削弱了。

Paul 和 Rúnar 所做的是简单地将 Scala 作为纯函数式编程语言。可变变量、异常、经典的输入输出和所有其他的非纯函数被消除了。假如你想知道在没有这些便捷方式下如何编写有用的代码，你需要阅读此书。从第一个原理扩展到增量的输入输出，本书展示了如何使用纯函数表达每一个概念。而且不仅仅是展示了可能性，也同样引导你去编写优美的代码和深入探索计算的本质。

本书是充满挑战的，不仅仅是因为它需要对细节的注意，同样是对你编程思想的挑战。通过阅读本书和完成推荐的练习，你将更好地认识纯函数式编程是什么，能表达什么，优点是什么。

本书让我特别喜欢的是它的自成体系。它开始于最简单的表达式，然后从细节解释每个抽象，再在其基础上进一步抽象。在某种程度上，本书开发了另一个 Scala“宇宙”，这里可变状态是不存在的，所有函数是纯的。普遍使用的 Scala 库的实现和这有些偏离，通常它们是部分按照命令式实现的，（大多数）外层是函数式接口。Scala 容许在函数式接口中封装可变状态，我认为这是一个优点。但是这种能力通常也被滥用。假如发现自己过多地使用它，那么本书是一种强力的解药。

MARTIN ODERSKY

Scala 的创造者

序言

编写好的软件很难。在各种方法论中纠结多年，我们俩发现并爱上了函数式编程（FP）。尽管它与众不同，但它就是能引领我们编写出一致连贯、灵活组合、美丽优雅的程序。

我们俩都是波士顿地区 Scala 爱好者群（Boston Area Scala Enthusiasts）的成员，这个群会定期在剑桥聚会。起初，群里主要是一些 Java 程序员，他们一直寻求一些更好的东西。后来大部分的人都表示，没有一个好的方法去学习如何用 Scala 进行函数式编程。我们学习的过程几乎都很随意，写一些函数式的代码，向其他 Scala 和 Haskell 程序员请教学习，阅读一些文章、博客和书籍。我们始终觉得应该有比这更简单的学习方法，直到 2010 年 4 月，群的组织者之一 Nermin Šerifović，建议我们写一本关于 Scala 函数式编程的书。本以为基于我们学习的经验，写一本期望中思路清晰的书是一件又快又容易的事情，没想到我们花了 4 年多才完成。要是我们当初学习函数式编程时，有这样一本书该多好啊。

希望这本书能够带给你一种兴奋刺激的感觉，犹如我们第一次遇到函数式编程那样。

致谢

这里我们要感谢很多曾经参与本书编写过程的人。首先是 Nermin Šerifović，我们波士顿 Scala 群的好朋友，没有他，我们不会开始写这本书。

然后是 Capital IQ 这个优秀的团队，感谢你们的支持，感谢你们自告奋勇地帮助测试了本书课程的第一个版本。

特别感谢 Tony Morris，他在本书早期阶段上的工作仍然非常有价值，尤其是在 Scala 函数式编程的实践上做出的贡献。

还有 Martin，感谢他为本书撰写的精彩前言，更感谢他创造了如此强大的编程语言并改变了整个行业。

感谢一直以来社区里那些热衷于函数式编程的 Scala 用户的鼓励。对于本书的审阅者、MEAP 的读者，以及每个为本书提供反馈建议的人，我们同样感谢你们，没有你们就没有这本书今天的样子。

还要尤其感谢开发编辑 Jeff Bleiel、图形编辑 Ben Kovitz、技术校对 Tom Lockney，以及每一个让本书变得更好的 Manning 的工作人员，包括以下开发阶段稿件的审阅者：Ashton Hepburn、Bennett Andrews、Chris Marshall、Chris Nauroth、Cody Koeninger、Dave Cleaver、Douglas Alan、Eric Torreborre、Erich W. Schreiner、Fernando Dobladez、Ionut、G. Stan、Jeton Bacaj、Kai Gellien、Luc Duponcheel、Mark Janssen、Michael Smolyak、Ravindra Jaju、Rintcius Blok、Rod Hilton、Sebastien Nichele、Sukant Hajra、Thad Meyer、Will Hayworth 和 William E. Wheeler。

最后，感谢 Sunita 和 Allison，正是因为你们一直以来的支持，才让这历经数年的冒险路程得以圆满。

关于本书

这不是一本关于 Scala 的书。这是一本介绍函数式编程（FP）的书——一种彻底的原则性很强的编写软件的方法。我们使用 Scala 作为交通工具到达那里，但是你可以在课程中使用任何其他编程语言。当你读完这本书，我们的目标是让你牢固地掌握函数式编程的概念，在写纯函数式程序时感到舒适，并能够为这些主题吸收新的素材，超越本书所覆盖的这些内容。

这本书的结构

这本书由四部分组成。在第一部分，我们讨论究竟什么是函数式编程，并引入一些核心概念。在第一部分的章节里做一个基础的技术总览，例如怎么组织小的函数式程序、定义纯函数式数据结构、错误处理，以及怎么和状态打交道。

在这基础之上，第二部分是一系列的函数式设计教程。我们通过一些实际的函数式库里的例子，揭露设计它们的思维过程。

通过第二部分的库函数的练习，会让你清楚这些库遵循某种模式并有一些冗余。这将突出我们对一种新的更高阶的抽象方式的需要，高阶的抽象能够写出更加泛化的库函数，我们将在第三部分引入这些抽象。这些都是对你的代码推导非常具有威力的工具。一旦你掌握它们，会让你成为一个极其富有生产力的程序员。

第四部分则填补了面向真实世界应用的其余部分的空缺，这些部分包括执行 I/O（例如写数据库、文件或视频显示器）、使用可变状态等，所有这些都是以纯函数式的风格来实现的。

在本书的学习过程中，我们依赖大量的编程练习，仔细、有序的练习能够帮助你消化这些资料。要理解函数式编程，不仅仅是学习理论，你必须立即打开你的文本编辑器写一些代码，你必须将那些你已经学习的理论，转换为工作中的实践。

我们还提供了所有章节的在线笔记。每一章都有一段与之相关的讨论，里面提供了一些进一步阅读的材料链接。这些章节笔记是可以被社区的读者们扩展的，是一个可编辑的 wiki：https://github.com/fpinscala/fpinscala/wiki。

受众人群

本书适合至少有一些编程经验的人参考阅读。在我们心中有一些特定种类的读者——那些对函数式编程好奇的，并且达到中级水平的 Java 或 C 程序员。但我们相信这本书也适合任何语言和任何经验水平的程序员。

在读这本书时你以前的专长经验并不一定比你的动机和好奇心更重要。函数式编程充满乐趣，但它也是一个有挑战性的课题，尤其是对于那些更有经验的程序员来说，因为它的思维方式可能不同于你以往有过的经验。不管你编程多久，你必须准备以一个初学者的姿态来学习。

读这本书并不需要之前有任何 Scala 经验，但我们不会花大量的时间和篇幅讨论 Scala 的语法和语言特性。我们将按照我们的需要，以最小化的方式来介绍 Scala，内容主要涉及的是其他素材。这些对 Scala 的介绍应该足以让你开始练习。如果你对 Scala 语言有进一步的问题，你应该用另一本 Scala 的书作为你的补充阅读（http://scala-lang.org/ documentation/books.html），或者在 Scala 语言文档（http://scala-lang.org/documentation/）里查找具体的问题。

如何阅读这本书

你可以按序阅读这本书，这四个部分的顺序也是有意设计的，你可以按照自己舒适的方式在这些部分之间先休息一下，把你所学的应用到工作中，然后再回来继续阅读下一部分。例如，第四部分的内容，只有你阅读完第一、二、三部分，并且非常熟悉函数式风格的编程之后才有意义。在第三部分之后，休息一下，尝试做一些比章节里更多的函数式程序练习，或许是个好主意。当然，最终都取决于你自己。

这本书里大部分章节都有相似的结构。先引入一些新的思想或技术，用例子来解释，然后通过一些练习来掌握。我们强烈建议你下载练习的源代码，在学习每一章的时候做做练习。练习、提示以及答案都可以在这里获取：https://github.com/fpinscala/fpinscla。我们也鼓励你访问 scala-functional Google 用户组（https://groups.google.com/forum/#!topic/scala-functional/），以及在 irc.freenode.net 上的 #fp-in-scala IRC 频道参与讨论。

练习题会根据难度和重要性来标记。我们会按照自己的理解来标记题目是否很难，或者它对章节材料的理解是否可选。带有困难标记的，我们会尝试给你一些期待的思路——这只是我们所猜测的，或许你会发现一些没有标记困难的问题很难解决，一些标记为困难的问题反而很容易。带有可选标记的练习主要为增进你的见识，在不妨碍你阅读后续的内容时你可以跳过。这些练习用下面的图标表示是否可选：

◈ **练习 1**

在一个被填充的四边形图标后边的练习表示它们是关键的练习。

◇ **练习 2**

一个空心的四边形表示练习是可选的。

这本书会贯穿很多例子，这些例子你要去尝试实践，而非仅仅只是阅读。在你开始之前，应该确保 Scala 解释器能运行。我们鼓励你用自己的方式去试验一下你看到的各种例子。帮你把事情搞明白的一个方法就是对它稍作改造，看看所做的改变对输出结果有什么影响。

有时我们会用 Scala 解释器会话来演示一些代码的运行或求值结果。这些将以解释器的提示符 scala> 开头。在提示符后边紧跟的是要输入或粘贴到解释器的代码，而下一行则是解释器的响应，就像这样：

```
scala> println("Hello, World!")
Hello, World!
```

代码规范和下载

在列表或文本中的所有代码都是*类似这样的*等宽字体，以便于把它们和普通文本区分开。Scala 里的关键字使用*类似这样的*等宽**粗体**。很多清单里都会包含代码注释，用于突出重要概念。

要下载这本书里的样例代码，以及练习题和章节注释里的代码，请到 https://github.com/fpinscala/fpinscala 或者出版商网站：www.manning.com/FunctionalProgramminginScala。

设定预期

虽然函数式编程对我们编写软件过程中的每个层面都产生了深远的影响，但它仍需要时间。这是一个渐进的过程。在第 1 章先不要指望能被神奇的函数式编程所惊艳到。安排在开头的一些原则都非常微妙，甚至看起来跟常识没什么区别。如果你觉得“这个我在不知道函数式编程的时候已经能够做到了”，那就对了，这正是重点所在。大多数程序员已经在某种程度上做过函数式编程的事情，只是他们不知道而已。大多数人认为最佳实践的很多东西（例如，让函数只具有单一职责，或让数据不可变）都受到函数式编程的启发。我们只是发现在这些最佳实践下面的原则，找出它们的逻辑结论。

在读这本书的时候有很大的可能你将同时要学习 Scala 语法和函数式编程。一开始这些代码可能看起来很陌生，这些技术也显得不太自然，并且练习题也很费脑子。这很正常，不要被吓到。如果你被问题卡住，看一下提示和答案，或把问题抛到 Google 用户组（https://groups.google.com/forum/#!topic/scala-functional/），或 IRC 频道（#fp-in-scala on irc.freenode.net）。

总之，我们希望这本书对你来说是有趣的、有益的体验，函数式编程能让你的工作更轻松、更愉悦，正如它已经带给我们的收益。这本书的目的是当你把里面所有说过的都尝试过，能帮助你在工作时更有效率。它应该让你感到不像是在写一些丑陋、不符合设计原理且难维护的软件，而更像是要创建一个美观实用的东西。

第一部分

函数式编程介绍

我们以一个激进的前提开始读这本书——限制自己只用纯函数来构造程序，纯函数是没有副作用的，比如读取文件或修改内存时。这种函数式编程的理念，或许非常不同于你以往的编程方式。因此，我们从头开始，重新学习如何用函数式风格来写一些简单的程序。

在第 1 章，我们将解释到底什么是函数式编程，并给你一些有益的理念。第一部分剩余部分的章节介绍了 Scala 函数式编程基本的技术。第 2 章介绍 Scala 语言，涵盖了一些比如怎样函数式地实现一个循环、怎样像普通值一样来操作函数等基础知识。第 3 章处理可能随时间变化的内存数据结构。第 4 章讨论用纯函数来处理错误。第 5 章将介绍非严格求值（non-strictness）的概念，它可以用于提升功能代码的效率和实现模块化。最后，第 6 章介绍使用纯函数来对状态建模。

本书第一部分的意图是让你纯粹按照函数层面的输入输出考虑程序，并教会你一些在第二部分开始写一些实践代码时所需要的技术。

1 什么是函数式编程

函数式编程（FP）基于一个简单又蕴意深远的前提：只用*纯函数*来构造程序——换句话说，函数没有*副作用*（side effects）。什么是副作用？一个带有副作用的函数不仅只是简单地返回一个值，还干了一些其他事情，比如：

- 修改一个变量
- 直接修改数据结构
- 设置一个对象的成员
- 抛出一个异常或以一个错误停止
- 打印到终端或读取用户的输入
- 读取或写入一个文件
- 在屏幕上绘画

我们将在本章对*副作用*提供一个更清晰的定义。考虑一下如果程序没有能力做这些事情，或处理这些行为时有显著的限制的话是什么样子？或许很难想象，这甚至根本写不出有用的程序。如果我们不能对一个变量重新赋值，怎么来写一个类似循环的程序？怎样处理数据的变化或不通过抛出异常的方式来处理错误？我们怎样才能写一个必须执行 I/O 的程序，例如在屏幕绘画或读取一个文件？

答案是函数式编程限制的是*怎样*写程序，而非表达*什么样*的程序。通过本书我们将学习到如何没有副作用地表达我们的程序，包括执行 I/O、处理错误、修改数据。我们将学习到为什么遵循函数式编程的规范是极其有益的，因为用纯函数编程更加模块化。由于纯函数的模块化特性，它们很容易被测试、复用、并行化、泛化以及推导。此外，纯函数减少了产生 bug 的可能性。

在这一章，我们将演示一个简单的带有副作用的程序，并通过去除这些副作用来示范函数式的一些好处。我们还会更广泛地讨论一些函数式的好处，并定义两个重要的概念——*引用透明*和*代换模型*。

1.1 函数式编程的好处：一个简单的例子

让我们通过一个例子示范纯函数编程的一些好处。这里主要是阐明一些基本理念，这也是整本书的要点。这可能是你第一次接触 Scala 语法。我们将在下一章讲述 Scala 语法的更多内容，所以别担心下面代码的每个细节，只要你大致明白这段代码在做什么就行。

1.1.1 一段带有副作用的程序

假设我们要为一家咖啡店的购物编写一段程序，先用一段带有副作用的 Scala 程序（也称作不纯的程序）来实现。

示例 1.1 一段带有副作用的 Scala 程序

```
class Cafe {

  def buyCoffee(cc: CreditCard): Coffee = {
    val cup = new Coffee()
    cc.charge(cup.price)
    cup
  }
}
```

类似于java，关键字class表示一个类，类体用大括号包围起来。

关键字def表示一个方法。cc: CreditCard定义了参数名cc和类型CreditCard。在参数列表之后的Coffee是buyCoffee方法的返回值类型。等号之后的花括号里的代码块是方法体。

不需要分号，语句之间通过换行符来界定。

副作用。信用卡计费。

我们不需要声明return，因为cup是最后一条语句，会自动return。

`cc.charge(cup.price)` 这行是一个副作用的例子。信用卡的计费涉及与外部世界的一些交互——假设需要通过 web service 联系信用卡公司、授权交易、对卡片计费，并且（如果前边执行成功）持久化一些记录以便以后引用。我们的函数只不过返回一杯咖啡，这些其他行为也额外（on the side）发生了，因此也被称为“副作用”（我们将在本章后边正式定义副作用）。

副作用导致这段代码很难测试。我们不希望测试逻辑真的去联系信用卡公司并对卡片计费。缺乏可测试性预示着设计的修改：按理说 `CreditCard` 不应该知道如何联系信用卡公司实际执行一次计费，同样也不应该知道怎么把一次计费持久化到内部系统。我们可以让 `CreditCard` 忽略掉这些事情，通过传递一个 `Payments` 对象给 `buyCoffee` 函数，使代码更加模块化和可测化。

示例 1.2 添加一个支付对象

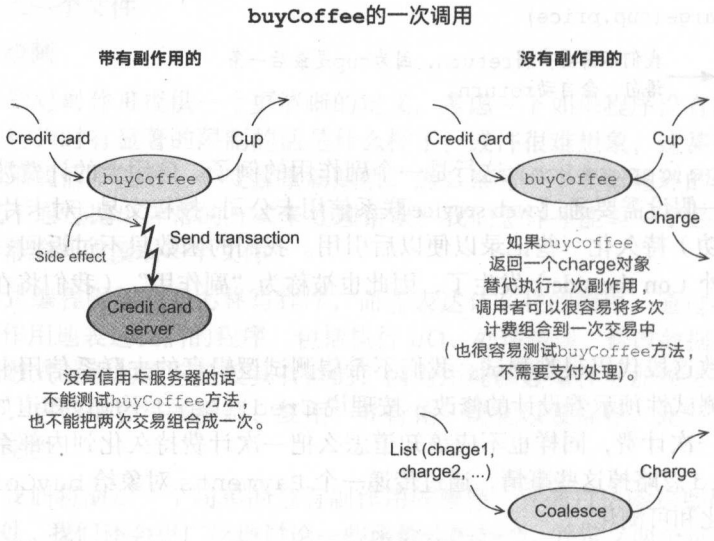
```
class Cafe {
  def buyCoffee(cc: CreditCard, p: Payments): Coffee = {
    val cup = new Coffee()
```

```
        p.charge(cc, cup.price)
        cup
    }
}
```

虽然当我们调用 p.charge(cc, cup.price) 的时候仍然有副作用发生，但至少恢复了一些可测试性。Payments 可以是一个接口，我们可以写一个适合于测试的 mock 实现这个接口。但这也不够理想，即便用一个具体类可能更好，我们也不得不让 Payments 成为一个接口，否则，任何 mock 都很难被使用。举个例子，在调用完 buyCoffee 之后或许我们要检测一些内部状态，我们的测试不得不保证这些状态在调用 charge 后已经被适当修改。虽然可以使用一个 mock 框架或相似的东西来为我们处理这些细节，但这样有些太小题大做了，我们只不过想测试一下 buyCoffee 产生的计费是否等于这杯咖啡的价格而已。

撇开对测试的担心，这里还有另一个问题：buyCoffee 方法很难被复用。假设一个叫 Alice 的顾客，要订购 12 杯咖啡。最理想的情况是只要复用这个方法，通过循环来调用 12 次 buyCoffee。但是基于当前的程序，会陷入 12 次对支付系统的调用，对 Alice 的信用卡执行 12 次计费！那样所产生的更多的手续费对 Alice 和咖啡店来说都不好。

多次购买的话我们该怎么做？如下图所示，我们可以编写一个全新的函数 buyCoffees，它使用特别的逻辑去批量处理计费。[1] 这里可能不是什么大问题，因为 buyCoffee 的逻辑很简单，但是在别的情况下我们需要重复的逻辑就会很多，而且要为失去代码的可复用性和可组合性而难过。



1 也可以写一个特定的 BatchingPayments 批量支付，实现 Payments 接口。它会以某种方式尝试批量逐次对同一张信用卡计费，不过这也会导致程序变得复杂。多少笔计费做一次批处理？应该等待多久？要强制 buyCoffee 方法表示批处理已结束的话或许需要调用 closeBatch 方法？如何在恰当的时候去做？

1.1.2 函数式的解法：去除副作用

函数式的解法是消除副作用，通过让 buyCoffee 方法在返回咖啡（Coffee）的时候把费用（Charge）也作为值一并返回。计费的处理包括发送到信用卡公司、持久化这条记录等，这些过程将在其他地方来做。下面是这段函数式解法的大致样子（先别太注重 Scala 的语法，下一章会详细讲解）：

```
class Cafe {
    def buyCoffee(cc: CreditCard): (Coffee, Charge) = {
        val cup = new Coffee()
        (cup, Charge(cc, cup.price))
    }
}
```

buyCoffee方法返回一对儿包含Coffee和Charge的值，使用类型(Coffee,Charge)来表示。这里不涉及支付相关的任何处理。

用小括号创建一个cpu和Charge的数据“对”，之间用逗号分隔。

我们把费用的创建过程跟它的执行过程分离。buyCoffee 函数现在的返回值里除了 Coffee 还有一个费用的值。我们很快就会看到如何更方便地在一个事务里复用这个函数来买多杯咖啡。不过先看看“Charge”怎么定义？我们所构造的这个数据类型由信用卡（CreditCard）和金额（amount）组成，并提供了一个 combine 函数，以便对同一张信用卡合并费用：

```
case class Charge(cc: CreditCard, amount: Double) {

    def combine(other: Charge): Charge =
        if (cc == other.cc)
            Charge(cc, amount + other.amount)
        else
            throw new Exception("Can't combine charges to different cards")
}
```

case类只有一个主构造器，构造参数紧跟着类名后边（这里类名是Charge）。这个列表中参数都是public的、不可修改的，访问时使用面向对象的方式，中间用点来标注，如other.cc。

if表达式跟java里的语法一样，但它也会返回其中一个分支结果的值。

如果cc == other.cc，那么combine将返回Charge(..)；否则在else分支会抛出异常。

抛出异常的语法也和Java或其他语言相似。我们在后续章节将会讨论更多用函数式来处理错误条件的方法。

case类可以不通过new关键字创建。我们只需要在类名后面传递一组参数到主构造器。

现在我们来看看 buyCoffees 方法如何实现购买 n 杯咖啡。与之前不同，现在可以利用 buyCoffee 方法实现，这正是我们所希望的。

示例 1.3 使用 buyCoffees 方法购买多杯咖啡

```
class Cafe {

    def buyCoffee(cc: CreditCard): (Coffee, Charge) = ...
```

List.fill(n)(x)创建一个对x复制n份的列表。我们将在后续章节解释这个有趣的函数调用语法。

List[Coffee]是一个承载Coffee的不可变的单向链表。我们将在第3章讨论数据类型的更多细节。

```
def buyCoffees(cc: CreditCard, n: Int): (List[Coffee], Charge) = {
  val purchases: List[(Coffee, Charge)] = List.fill(n)(buyCoffee(cc))
  val (coffees, charges) = purchases.unzip
  (coffees, charges.reduce((c1,c2) => c1.combine(c2)))
}
```

unzip将数值对儿列表，分成一对儿(pair)列表。这里我们用一行代码对这个pair解构成2个值(coffee列表和charge列表)。

charges.reduce对整个charge列表规约成一个charge，每次使用combine来组合两个charge。reduce是一个高阶函数的例子，我们将在下一章做适当的介绍。

总体上看，这个解决方案有显著的改善，现在我们可以直接复用 buyCoffee 来定义 buyCoffees 函数，这两个函数都很简单并容易测试，不需要实现一些 Payments 接口来进行复杂的 mock！事实上，Cafe 现在完全忽略了计费是如何处理的。当然我们可以用一个 Payments 类来做付款处理，但 Cafe 并不需要了解它。

让 Charge 成为一等（first-class）值，还有一些我们没有预期到的好处：我们能更容易地组装业务逻辑。举个例子，Alice 带着她的笔记本电脑来咖啡店并在这里工作几个小时，中间时不时会买（几杯）咖啡。如果咖啡店能够把 Alice 买的咖啡合并成一笔费用以节约信用卡的手续费就太好了。因为 Charge 是一等值，我们可以用下面的函数把同一张信用卡的费用合并为一个 List[Charge]：

```
def coalesce(charges: List[Charge]): List[Charge] =
  charges.groupBy(_.cc).values.map(_.reduce(_ combine _)).toList
```

我们像传值一样传递函数给 groupBy、map 和 reduce 方法。在接下来的几章，你将学会读和写类似这样的一行程序。_.cc 和 _ combine _ 是匿名函数的语法，在后边的章节会讲到。

你或许因为代码里的符号过于紧凑而感到难以理解，这本书将很快让你习惯读写这样的 Scala 代码。这个函数接收一个计费列表参数，按照信用卡对这个参数进行 group，然后对每张卡组合成一个单独的计费。这样完美地实现了复用性和可测试性，不需要加任何额外的 mock 对象或接口。想象一下用我们的第一个 buyCoffee 来实现相同的逻辑。

这只是为了阐述函数式有哪些它所声称的好处，所以故意采用一个简单的例子来说明。好的重构通常看起来很自然、不起眼，就像是一种标准做法。函数式编程把各种想法规约成逻辑表达，这种规约对一些适用性不是那么明显的情况也适用。正如你通过这本书将要学习到的，持续遵循函数式编程原则的意义深远，并会有巨大的收益。函数式编程对于程序在每个层面组合——从简单的循环到高层架构，是一种真正的转变。它所呈现的风格是完全不同的，希望你能欣赏这种漂亮、紧凑的编程方式。

现实世界是如何的?

我们通过 buyCoffee 这个例子看到如何把计费的创建过程与处理过程分离。总的来说，就是通过把这些副作用推到程序的外层，来转换任何带有副作用的函数。

对函数式程序员而言，程序的实现，应该有一个纯的内核和一层很薄的外围来处理副作用。

但即便如此，某些时候我们不得不面对现实情况产生的副作用，并通过一些外部系统提交费用处理。难道没有其他需要副作用或变化（mutation）的可用的程序？我们怎样写这样的程序？通过本书，我们将发现很多看似必须存在副作用的程序都有对应的函数式实现。如果不存在对应的函数式实现，我们会找到一种方式构造代码让副作用发生但不可见（例如，可以在一些函数体里声明局部变量，或者当没有外围函数可以观察到这种情况时，写入一个文件）。

1.2 （纯）函数究竟是什么

我们之前说过函数式编程意味着使用纯函数，纯函数是没有副作用的。在咖啡店的例子里，给出了副作用和纯函数的非正式的概念。这里正式化这个概念，更精确地定义函数式编程。这样我们可以更深入地了解函数式编程的一个好处：纯函数更容易推理。

对于一个输入类型为 A 和输出类型为 B 的函数 f（用 Scala 写为：`A => B`，读作“A 到 B”或“A 箭头 B”），它是一种将所有的 A 类型的值 a 关联到某个确切的 B 类型的值 b 的运算过程，即 b 完全由 a 来决定。任何内部或外部过程的状态改变都不会影响到 `f(a)` 的计算结果。例如，函数 `intToString` 的类型为 `Int=>String`，它负责将整数转换为一个相应的字符串。除此之外应该什么也不用做。

换言之，一个函数在程序的执行过程中除了根据输入参数给出运算结果之外没有其他的影响，就可以说是没有副作用的。我们有时更加明确地把这一类函数称为“纯函数”，但这有点多余，除非我们额外声明，否则函数就是指那些没有副作用的。[2]

你可能已经了解过一些纯函数了，比如考虑整数的加法（+）函数。它接收两个整数值并返回一个整数值，对于给定的两个整数值，它的返回值永远是相同的整数值。另一个例子是 Java、Scala 和其他将字符串作为不可变对象的语言中对字符串求长度（`length`）的函数，对于给定的字符串，返回的长度总是相同的。

我们可以使用引用透明（referential transparency）的概念对纯函数进行形式化。这不仅仅是函数的属性，而且是一般表达式的属性。为了便于讨论，设定一个程序的任何部分的表达式，都可以计算为一个结果——在 Scala 解释器下输入的任何内容都会得到结果。例如，`2 + 3` 是一个表达式，它使用纯函数 + 对数值 2 和 3 进行运算（注意，数值 2 或 3 同样也是一个表达式）。它没有副作用，这个表达式每次运算的结果都是同一个数值 5。实际上，如果看到程序里有 `2 + 3` 这样的表达式可以直接替换为 5，并不会改变程序的含义。

这意味着任何程序中符合引用透明的表达式都可以由它的结果所取代，而不改变该程序的含义。当调用一个函数时所传入的参数是引用透明的，并且函数调用也是引用透明的，那么这个函数是一个纯函数。接下来会介绍一些例子。

2 Procedure（过程）经常用于表示带有副作用的参数化的代码块。

引用透明与纯粹度

对于程序 p，如果它包含的表达式 e 满足引用透明，所有的 e 都可以替换为它的运算结果而不会改变程序 p 的含义。假设存在一个函数 f，若表达式 f(x) 对所有引用透明的表达式 x 也是引用透明的，那么这个 f 是一个纯函数。[3]

1.3 引用透明、纯粹度以及替代模型

让我们看看是如何对原先的 buyCoffee 例子定义为引用透明的：

```
def buyCoffee(cc: CreditCard): Coffee = {
    val cup = new Coffee()
    cc.charge(cup.price)
    cup
}
```

无论 cc.charge(cup.price) 返回什么类型（也许是 Unit，相当于其他语言的 void），它都会被 buyCoffee 丢弃。

因此，buyCoffee(aliceCreditCard) 的运算结果将仅仅是一杯咖啡，这相当于 new Coffee()。根据我们定义的引用透明，如果 buyCoffee 要满足纯函数，对任何 p 而言，p(buyCoffee(aliceCreditCard)) 行为需要与 p(new Coffee()) 相同。这显然不成立，表达式 new Coffee() 不做任何事，而 buyCoffee(aliceCreditCard) 将会连接信用卡公司并授权计费。两个程序显然有差异。

引用透明要求函数不论进行了任何操作都可以用它的返回值（value）来代替。这种限制使得推导一个程序的求值（evaluation）变得简单而自然，我们称之为*代替模型*（substitution model）。如果表达式是引用透明的，可以想象计算过程就像在解代数方程。展开表达式的每一部分，使用指示对象替代变量，然后归约到最简单的形式。在这一过程中，每一项都被等价值所替代，计算的过程就是被一个又一个等价值（equal）所替代的过程。换句话说，引用透明使得程序具备了*等式推理*（equational reasoning）的能力。

让我们再看两个例子，其中一个例子的所有表达式都是引用透明的，可用替代模型来推导，另一个例子则违反了引用透明。这并不复杂，只是对你可能已经理解的东西进行形式化。

让我们尝试在 Scala 解析器（也称作 REPL，发音类似“ripple”，把其中的 i 换为 e）下执行下面的代码。注意，在 Java 和 Scala 中，字符串是不可变的。一个修改过的字符串其实是一个新的字符串，老的字符串保持不变：

```
scala> val x = "Hello, World"
x: java.lang.String = Hello, World
```

3 这个定义有点模糊，我们会在本书后边再提炼它。更多讨论可以看我们在 github 网站的这一章的笔记（https://github.com/pchiusano/fpinscala）。

```
scala> val r1 = x.reverse
r1: String = dlroW ,olleH

scala> val r2 = x.reverse ←——r1和r2相同。
r2: String = dlroW ,olleH
```

假设我们把所有使用 x 的地方用它所引用（定义）的表达式来替换：

```
scala> val r1 = "Hello, World".reverse
r1: String = dlroW ,olleH

scala> val r2 = "Hello, World".reverse ←——r1和r2依然相同。
r2: String = dlroW ,olleH
```

这样做并不影响结果。r1 和 r2 的值和以前一样，所以说 x 是引用透明的。另外，r1 和 r2 也同样是引用透明的，如果它们出现在某个程序中也可以替代为它们所引用的值而不会对程序造成影响。

现在来看看一个不是引用透明的函数。比如 java.lang.StringBuilder 类里的 append 方法，这个方法改变了 StringBuilder 对象自身。在调用 append 方法之后 StringBuilder 对象之前的状态被破坏：

```
scala> val x = new StringBuilder("Hello")
x: java.lang.StringBuilder = Hello

scala> val y = x.append(", World")
y: java.lang.StringBuilder = Hello, World

scala> val r1 = y.toString
r1: java.lang.String = Hello, World

scala> val r2 = y.toString
r2: java.lang.String = Hello, World ←——r1和r2相同。
```

到目前为止一切还好，现在看看破坏引用透明所引起的副作用。假设我们像之前一样替代 append 调用，把 y 替换为它所引用的表达式：

```
scala> val x = new StringBuilder("Hello")
x: java.lang.StringBuilder = Hello

scala> val r1 = x.append(", World").toString
r1: java.lang.String = Hello, World

scala> val r2 = x.append(", World").toString
r2: java.lang.String = Hello, World, World ←——r1和r2不再相同。
```

这一替换导致了不同的结果，所以我们断定 StringBuilder.append 不是一个纯函数。虽然 r1 和 r2 看上去是同一个表达式，但它们引用的 StringBuilder 是两个不同的值。在 r2 调用 x.append 的时候，r1 已经改变了 x 引用的对象。正是这个原因让程序看上去难以理解，副作用使程序行为的推理更加困难。

与之相反的是，替代模型则很容易推理，因为对运算的影响纯粹是局部的（只对那些赋值表达式产生影响），不需要先在内心模拟一系列状态的更新才理解这一段代码。只需要理解*局部的推理*（local reasoning），不必费心地去跟踪函数执行前后的状态变化，只用简单地看一下函数的定义，把它替换成一个参数。即使没有用过“替代模型”这一名词，你也一定用过这种方式来推理程序。[4]

对纯粹度的概念进行形式化，有助于我们深入了解函数式编程为什么更加模块化。模块化程序是由组件组成的，这些组件又是可理解、可复用，并独立于整体存在的。整体程序只取决于组件和它们组成的规则，也就是说它们是可组合的（composable）。纯函数是模块化的、可组合的，因为它从“对结果做什么”和“如果获取输入”中分离了计算本身的逻辑，就像一个黑盒子。对输入的获取只有一种方式：通过参数传给函数。输出也只是简单地将计算结果返回。把这些关注点分离开，计算也更容易被复用。我们可以复用这些逻辑，而不必担心输入或输出对整个上下文引起的副作用。可以看到在 `buyCoffee` 的例子里，消除了支付完成时的输出所引起的副作用，测试或者进一步组合（比如 `buyCoffees` 和 `coalesce`）时，函数更容易复用。

1.4 小结

本章介绍了函数式编程并解释了函数式编程到底是什么，为什么要使用它。通过一些简单的例子来说明函数式编程的好处，随着课程的深入你会对函数式编程的优点更加清晰。还对*引用透明*和*代替模型*进行了讨论，并解释了为何函数式编程能够更易于推导，更加模块化。

本书从最简单的任务开始，你将学习到适用于各个层面的函数式编程的概念和原则。在接下来的章节里将介绍一些基础——如何用函数式编程写一段循环，或实现一个数据结构，以及如何处理错误和异常。我们需要学习如何做这些事情并能自如地运用一些低级别的函数式惯用写法。我们在第二、三、四部分对函数式设计的探讨，需要建立在这一理解之上。

4 在实践中，程序员不需要花时间机械地应用替换来判定代码是不是纯粹的，通常这种判断是很显而易见的。

2 在 Scala 中使用函数式编程

现在我们致力于使用纯函数，那么问题来了：怎样才能写出哪怕是最简单的一个程序？我们习惯于把程序当成序列按顺序来执行指令，每一个指令都会产生某种副作用。在这一章，我们开始学习如何通过组合纯函数来写 Scala 程序。

这一章主要面向刚接触 Scala 或函数式编程的读者。学习陌生语言的一种有效方式是沉浸其中，所以我们需要跳进去全心投入。唯一不让 Scala 代码看上去那么陌生的方式就是阅读大量的 Scala 代码。在第 1 章我们已经看过一些了，在这一章我们开始看一些比较小的但完整的程序。我们会把它们分解为一小段一小段来解释它的细节，好让我们理解 Scala 语言的基础语法。

一旦掌握了 Scala 语言的一些基本元素，便可以引入一些函数式编程的基本技术。我们会讨论如何使用*尾递归函数*（tail recursive function）来写一段循环逻辑，也会引入*高阶函数*（HOF）。高阶函数可以接收一个函数作为参数或者将函数作为它的返回值输出。我们也会看一些*多态*（polymorphic）高阶函数的例子，它们是由类型引导实现的多态[1]。

本章涉及很多新的内容。一些内容是与那些让你费解的高阶函数相关的，如果你以前用过的编程语言不能把函数像值一样传递的话。记住，这一章的关键不是去消化每一个概念或做所有的练习题，我们后续会从不同的角度回顾这些概念，这里只是一个初步印象。

2.1 Scala 语言介绍：一个例子

下面是一个我们要讨论的完整的 Scala 程序。这里不引入函数式编程的任何新的概念，只涉及 Scala 语言和语法。

示例 2.1 一个简单的 Scala 程序

```
// 一行注释
/* 另一行注释 */
/** 文档说明 */
object MyModule {  ←——声明一个单例对象，即同时声明一个类和它的唯一实例。
```

1 不同于继承和子类型多态，这里说的是类型参数化多态。——译者注

```
  def abs(n: Int): Int =  <——abs方法接收一个integer并返回一个integer。
    if (n < 0) -n  <——如果n小于0，返回-n
    else n

  private def formatAbs(x: Int) = {  <——一个私有方法只能被MyModule里的其他成员调用。
    val msg = "The absolute value of %d is %d"  <——字符串里有2个的占位符，%d代表数字。
    msg.format(x, abs(x))  <——在字符串里将2个%d占位符分别替换为x和abs(x)。
  }

  def main(args: Array[String]): Unit =  <——Unit类似于Java或C语言里的void。
    println(formatAbs(-42))
  }
}
```

我们声明了一个名为 MyModule 的单例对象（也称为模块）。只是简单地封装一段代码，随后可以通过名字来引用。Scala 代码必须放在一个单例对象（object）或一个类（class）中，出于简单，这里使用一个单例对象，把代码放在花括号之间。稍后会讨论单例对象与类之间的更多细节，现在我们只关注这个特定的单例对象。

object 关键字

object 关键字创建一个新的单例类型，就像一个 class 只有一个被命名的实例。如果你熟悉 Java，在 Scala 中声明一个 object 有些像创建了一个匿名类的实例。Scala 没有等同于 Java 中 static 关键字的概念，object 在 Scala 中经常用到，就像你在 Java 中用一个带有静态成员的 class 一样。

MyModule 对象有 3 个方法，它们使用 def 关键字声明：abs、formatAbs 和 main。我们使用术语“方法”来指代一些在单例对象或类的内部通过 def 关键字声明的函数或成员。现在我们逐一看一下 MyModule 里的方法。

abs 方法是一个接收整型类型参数并返回其绝对值的纯函数：

```
def abs(n: Int): Int =
  if (n < 0) -n
  else n
```

在 def 关键字之后紧跟着方法名，然后是小括号，里面是方法参数。在这个例子中，abs 方法只有一个 Int 类型的单个参数。在参数列表闭括号之后是一个可选的类型（:Int），它表示返回值是 Int 类型（前边的冒号表示存在某种类型）。

在等号（=）之后的部分是方法体。有时我们对等号前边的声明部分称为“左手边的”或“签名”，等号后边的部分称为“右手边的”或“定义”。注意，如果没有显式地使用 return 关键字，一个方法的返回值就是右手边的求值结果。所有的表达式，包括 if 表达式都产生一个结果。这里代码中“右手边的”只是简单地返回 -n 或者 n（取决于是否 n < 0）的表达式。

formatAbs 方法是另一个纯函数：

```
private def formatAbs(x: Int) = {
  val msg = "The absolute value of %d is %d."
  msg.format(x, abs(x))  ←——format标准库的String里的一个方法。
}
```

这里对 msg 对象调用 format 方法，传入了一个 x 值和对 x 运用过绝对值（abs）的值。结果是一个新的字符串，这个字符串里出现 %d 的地方被替代为了 x 和 abs(x)。

这个方法定义为 private，意味着无法在 MyModule 对象之外调用。它接收一个 Int 类型，返回一个 String。注意返回值类型没有声明，因为 Scala 有能力推断一个方法的返回值类型，所以我们省略了，但是通常考虑到与他人协作和保持良好的代码风格，最好还是显式地声明返回值类型。

方法体中包含了多条声明，所以把它们放在花括号中。一对花括号所包含的声明也称为“代码块”（block）。声明由分号或换行符分割。这个例子中，每个声明都在单独一行里，所以可以不用分号。

代码块的第一行声明使用 val 关键字定义了一个字符串，命名为 msg。只是简单地给一个名字以便引用。val 表示不可变变量，所以 formatAbs 方法体中 msg 会一直引用同一个字符串值。如果你在方法体中将 msg 重新赋值了一个值，编译器将会报错。

记住，一个方法只是简单地返回它右手边的值。所以不必需要 return 关键字。这个例子中右手边是一个代码块。在 Scala 中，一段包含在大括号里的多行声明（multistatement）代码块的值等于它最后一个表达式的返回值。因此，formatAbs 方法的值就是 msg.format(x, abs(x)) 的返回值。

最后，main 方法是一个调用纯函数内核的外壳，并打印结果到终端。有时我们称这样的方法为“过程”（或非纯函数）而非函数，以突出它们是带有副作用的。

```
def main(args: Array[String]): Unit =
  println(formatAbs(-42))
```

以 main 命名的方法是一个特殊方法。当程序运行时 Scala 会查找以 main 命名的特定签名的方法。它接收一个字符串数组作为参数，返回值类型是 Unit。

参数数组包含了从命令行传递过来的参数，我们在这里并没有使用。

Unit 与 C 或 Java 编程语言中的 void 有相似的目的。在 Scala 中只要没有引起程序崩溃或挂住，每个方法都会返回值。但 main 方法的返回值没有任何意义。它是一个特殊的类型 Unit，该类型只有唯一的值，文法上写为 ()，一对小括号，发音为“unit”。通常返回 Unit 类型的方法暗示它包含副作用。

main 方法体把调用 formatAbs 返回的结果字符串打印到终端，注意 println 方法的返回值是 Unit，也正是 main 方法需要返回值的类型。

2.2 运行程序

本节讨论如何用最简单的方式运行 Scala 程序，适合一些简短的例子。更典型的方式是使用 sbt（Scala 构建工具）来构建和运行 Scala 项目，并使用 IntelliJ 或 Eclipse 之类的 IDE 来开发。参考本书在 GitHub（https://github.com/fpinscala/fpinscala）上的代码来了解更多配置 sbt 的信息。

运行 Scala 程序（MyModule）的最简单方式是从命令行直接调用 Scala 编译器。先把代码放到一个名为 MyModule.scala 之类的文件里，然后使用 scalac 编译成 Java 字节码：

```
> scalac MyModule.scala
```

这将生成一些以 .class 后缀结尾的文件。这些文件包含可运行在 Java 虚拟机上的编译过的代码。该代码可以使用 Scala 的命令行工具来执行：

```
> scala MyModule
The absolute value of -42 is 42.
```

实际上，Scala 代码并不严格需要先通过 scalac 编译。像之前写的简单程序可以直接通过命令行传递给 Scala 解析器来运行：

```
> scala MyModule.scala
The absolute value of -42 is 42.
```

这在使用 Scala 处理脚本时很有用。解释器会从 MyModule.scala 文件的所有对象里寻找包含适当签名的 main 方法，并调用它。

最后，一个可选的方式是以交互的方式启动一个 Scala 解释器——REPL（read-evaluate-print loop）。在写 Scala 程序时开一个 REPL 窗口方便验证代码。

可以在 REPL 中加载源代码文件来验证（你的终端实际输出可能有所不同）：

```
> scala
Welcome to Scala.
Type in expressions to have them evaluated.
Type :help for more information.

scala> :load MyModule.scala
Loading MyModule.scala...
defined module MyModule

scala> MyModule.abs(-42)
res0: Int = 42
```

:load是REPL里的一个命令，用于加载并解释一个Scala源文件（注意，不幸的是它对带有包名的Scala文件不能运行）。

我们可以在提示符后边输入Scala的表达式。

REPL对Scala表达式求值，并打印结果。它同时也会把结果赋给一个名为res0的变量（后续可以引用这个变量），并会显示它的类型，这个例子里是Int类型。

也可以在 REPL 中复制粘贴一段代码。它甚至有一个专为粘贴多行代码设计的粘贴模式（以 :paste 命令开启）。最好熟悉 REPL 的功能，因为在用 Scala 编程的时候它是一个常用的工具。

2.3 模块、对象和命名空间

在这一节我们将讨论 Scala 语法中与模块、对象和命名空间有关的内容。在刚才的 REPL 会话中，为了引用 abs 方法，我们必须用 MyModule.abs 的方式，因为 abs 定义在 MyModule 对象中。MyModule 可以说是它的命名空间。抛开技术层面的细节，Scala 中的每一个值都可以当成一个对象（object），[2] 每个对象都有零个或多个成员。对象的主要目的是给成员一个命名空间，有时我们也称为模块。一个成员可以是一个以 def 关键字声明的方法，或另一个以 val 或 object 声明的对象。对象还可以有其他类型的成员，我们现在先忽略。

典型的面向对象语言中，访问对象中的成员使用“.”符号，也就是一个命名空间（所引用对象的名字）后边跟着一个点（英文的句号字符），再后边跟着成员的名字，比如 MyModule.abs(-42)。为了使用 42 这个对象的 toString 成员，我们写为 42.toString。对象里的成员可以不受限制地相互引用（不需要对象名前缀），但如果要访问外围对象（enclosing object），需要一个特殊的变量：this。[3]

注意，即使 2+1 这样的表达式也是调用对象成员。这里是对对象 2 调用其 + 方法成员。它是表达式 2.+(1) 的语法糖，把 1 传给对象 2 的 + 方法。Scala 没有操作符（operator）的概念。+ 在 Scala 里是一个方法。若方法只包含一个参数，可以使用中缀方式来调用，即省略了点和小括号。比如 MyModule.abs(42) 可以写为 MyModule abs 42，结果是一样的，用哪种方式都行。

可以将一个对象的成员导入到当前作用域，这样就可以不受约束地使用它们：

```
scala> import MyModule.abs
import MyModule.abs

scala> abs(-42)
res0: 42
```

也可以使用下画线语法来导入一个对象的所有成员（非私有）：

```
import MyModule._
```

2.4 高阶函数：把函数传给函数

现在掌握了 Scala 的一些基础语法，我们将继续学习一些函数编程基础。你需要了解的第一个新概念：*函数也是值*，就像其他类型的值，比如整型、字符串、列表；函数也可以赋值给一个变量、存储在一个数据结构里、像参数一样传递给另一个函数。

---

2 出于讨论的目的，像 Int 这样的基础类型的值也被认为是对象，不同于 Java。

3 注意在这本书里我们使用*函数*这个术语泛指那些所谓独立的函数，比如 sqrt 或 abs，或某些类里的成员，包括方法。如果清楚上下文，*方法*和*函数*这两个术语也可能互换。既然重要的不是语法怎么写（比如 obj.method(2) 与 method(obj,12)），那么我们实际所说的是参数化的代码块。

把一个函数当作参数传递给另一个函数在纯函数式编程里很有用，它被称为高阶函数（higher-order function，缩写为 HOF）。接下来将通过一些例子来阐明。在本章的后面，将看到这种方法的威力，以及它是如何影响函数式编程风格的。但一开始，先适配我们的程序打印出一个数的绝对值和另一数的阶乘。这是一个运行结果：

```
The absolute value of -42 is 42
The factorial of 7 is 5040
```

2.4.1 迂回做法：使用循环方式

首先，我们来写一个阶乘：

```
def factorial(n: Int): Int = {
    def go(n: Int, acc: Int): Int =
        if (n <= 0) acc
        else go(n-1, n*acc)

    go(n, 1)
}
```

一个内部函数，或局部定义函数。在Scala中把一个函数定义在另一个函数体内很常见。在函数式编程中，我们不应该认为它跟一个局部整数或局部的字符串有什么不同。

想不通过修改一个循环变量而实现循环功能，可以借助递归函数。我们在阶乘函数体内部定义了一个辅助的递归函数。这种辅助函数习惯上被称为 go 或 loop。在 Scala 中函数可以定义在任何代码块中，包括在另一个函数内部。就像一个局部变量，go 函数只能被 factorial 函数内部引用。factorial 函数的定义最终不过只是以循环的初始条件不断地调用 go。

传给 go 的参数是循环的状态。在这个例子中包含 n 和一个当前阶乘的累计值 acc。为进行下一次迭代，只需用新的循环状态递归调用 go 函数（这里是 go(n-1, n*acc)），要退出循环，返回一个不继续进行递归调用的值（这个例子中是 n <= 0）。Scala 会检测到这种*自递归*（self-recursion），只要递归调用发生在*尾部*（tail position），编译器优化成类似 while 循环[4]的字节码。可以参考注释里关于这种技术的细节。基本思路是当递归调用后没有其他额外调用时，会应用这种优化[5]（称为*尾调用消除*）。

Scala 中的尾调用

我们说的尾调用是指调用者在一个递归调用之后不做其他事，只是返回这个调用结果。比如之前讨论的递归调用 go(n-1, n*acc)，它是一个尾调用，因为它没有做其他事情，直接返回了这个递归调用的结果。另一种情况：1 + go(n-1,

4　我们可以在 Scala 里手动写 while 循环，但非得这样做的情况很少，这是一种不好的方式，因为对于更好的组合风格来说它是一种阻碍。

5　术语优化在这里并不恰当。优化通常意味着一些非本质的性能提升，但当我们用尾调用来写循环时，一般依赖它们在被编译为迭代的循环（iterative loop），这样在每次迭代时不消耗栈帧的调用开销（栈调用在输入很大的情况下可能会 StackOverflowError）。

n*acc)，这里 go 不再是尾部，因为这个方法的结果还要参与其他运算（即结果还要再与 1 相加）。

如果递归调用是在一个函数的尾部位置，Scala 会自动把递归编译为循环迭代，这样不会每次都进行栈的操作。默认情况下 Scala 不会告诉你尾调用是否消除成功，可以通过 tailrec 注释（http://mng.bz/bWT5）来告诉编译器，如果编译不能消除尾部调用会给出编译错误。语法如下：

```
def factorial(n: Int): Int = {
    @annotation.tailrec
    def go(n: Int, acc: Int): Int =
        if (n <= 0) acc
        else go(n-1, n*acc)
    go(n, 1)
}
```

关于注释（annotation）我们在这本书里不想谈论太多（可以登录 http://mng.bz/GK8T 了解更多信息），不过 @annotation.tailrec 注释将会被我们广泛使用。

注意，关于练习请参考本章序言里的信息。

◇ **练习 2.1**

写一个递归函数，来获取第 n 个斐波那契数，前两个斐波那契数 0 和 1，第 n 个数总是等于它前两个数的和——序列开始为 0、1、1、2、3、5。应该定义为局部（local）尾递归函数。

```
def fib(n: Int): Int
```

2.4.2 第一个高阶函数

现在我们已经有了阶乘函数，让我们编辑一下之前的程序把它也引入进去。

示例 2.2 一个简单的包含 factorial 函数的程序

```
object MyModule {
    ... <——这里是abs和factorial的定义。
    private def formatAbs(x: Int) = {
        val msg = "The absolute value of %d is %d."
        msg.format(x, abs(x))
    }

    private def formatFactorial(n: Int) = {
        val msg = "The factorial of %d is %d."
        msg.format(n, factorial(n))
    }
```

```
    def main(args: Array[String]): Unit = {
        println(formatAbs(-42))
        println(formatFactorial(7))
    }
}
```

formatAbs 和 formatFactorial 这两个函数几乎是相同的，可以将它们泛化为一个 formatResult 函数，它接收一个函数参数。

```
def formatResult(name: String, n: Int, f: Int => Int) = {
    val msg = "The %s of %d is %d."
    msg.format(name, n, f(n))
}
```

f必须是一个接收Int返回Int的函数。

formatResult 是一个高阶函数（HOF），它接收一个函数 f 为参数。我们给 f 参数声明一个类型，就像其他参数那样，它的类型是 Int=>Int（读作 int to int 或 int 箭头 int），表示 f 接收一个整型参数并返回一个整型结果。

abs 函数与这个类型匹配，它接收 Int 并返回 Int。同样，factorial 也与这个类型匹配。因此我们可以把 abs 或 factorial 当参数传给 formatResult：

```
scala> formatResult("absolute value", -42, abs)
res0: String = "The absolute value of -42 is 42."

scala> formatResult("factorial", 7, factorial)
res1: String = "The factorial of 7 is 5040."
```

变量命名约定

对于高阶函数的参数，以 f、g 或 h 来命名是一种习惯做法。在函数式编程中，我们倾向于使用短的变量名，甚至单个字母命名。因为高阶函数的参数通常没法表示参数到底执行什么，无法体现它们的含义。许多函数式程序员觉得段名称让代码更易读，因为代码结构第一眼看上去更简单。

2.5 多态函数：基于类型的抽象

目前我们定义的函数都是单态的（monomorphic）：函数只操作一种数据类型，比如 abs 和 factorial 的指定参数类型是 Int。高阶函数 formatResult 也是固定的操作 Int=>Int 类型的函数。通常，特别是在写高阶函数时，希望写出的这段代码能够适用于任何类型，它们被称为“多态函数”。[6] 这里先介绍一下它的概念，在本章的后边会有更多的应用。

6 这儿的术语多态与你以往所熟悉的面向对象编程里的多态稍微有些差异，面向对象里的多态通常意味着某种形式的子类型或继承关系。这个例子中没有接口或子类型。这里所用的多态形式有时也称为参数化多态。

2.5.1 一个多态函数的例子

我们经常是先注意到若干个单态函数有相似的结构，才发现应该用一个多态函数来解决问题。比如，下面的单态函数 findFirst 返回数组里第一个匹配到 key 的索引，或在匹配不到的情况下返回 -1。它是从一个字符串数组里查找一个字符串的特例。

示例 2.3 在数组中查找字符串的单态函数

```
def findFirst(ss: Array[String], key: String): Int = {
    @annotation.tailrec
    def loop(n: Int): Int =
        if (n >= ss.length) -1
        else if (ss(n) == key) n
        else loop(n + 1)
    loop(0)
}
```

如果n到了数组的结尾，返回-1，表示这个key在数组里不存在。

ss(n)抽取数组ss里的第n个元素。如果第n个元素等于key返回n，表示这个元素出现在数组的索引。

否则，传入n加1，继续查找。

从数组的第一个元素开始启动loop。

这段代码的细节不是我们关注的重点，重要的是不管是从 Array[String] 中查找一个 String，还是从 Array[Int] 中查找一个 Int，或从任何 Array[A] 中查找一个 A，它们看起来几乎都是相同的。我们可以写一个更泛化的适用任何类型 A 的 findFirst 函数，它接收一个函数参数，用来对 A 进行判定。

示例 2.4 在数组中查找元素的多态函数

```
def findFirst[A](as: Array[A], p: A => Boolean): Int = {
    @annotation.tailrec
    def loop(n: Int): Int =
        if (n >= as.length) -1
        else if (p(as(n))) n
        else loop(n + 1)

    loop(0)
}
```

用类型A做参数替代掉String类型的硬编码，并且用一个对数组里每个元素进行测试的函数替代掉之前用于判断元素是否与给定key相等的硬编码。

如果函数p匹配当前元素，就找到了相匹配的元素，返回数组当前索引值。

这是一个多态函数的例子，有时也称作“泛型函数”。我们对数组和用于查找的函数，基于类型进行了抽象化。要写一个多态函数，我们引入了一种使用逗号分隔的类型参数（type parameter），紧跟在函数名称后使用中括号括起来（这里是单个类型参数 [A]）。可以给类型参数起任何名字，比如 [Foo,Bar,Baz] 或 [TheParameter,another-good-one] 都是有效的类型参数名，不过习惯上通常用短的、单个大写的字母来命名类型参数，比如 [A,B,C]。

在类型参数列表中引入的类型变量，可在其他类型签名中引用（类似于参数列表中的参数变量可在函数体中引用）。在 findFirst 函数中类型变量 A 被两个地方引用：一处是数组元素要求是类型 A（声明为 Array[A]），另一处是函数 P 必须接收类型 A（声明为 A=>Boolean）。这两处类型签名中引用相同的类型变量，意味着它们的类型必须

相同。当我们调用 findFirst 时编译器会强制检测，如果在 Array[Int] 中查找一个 String，可能会造成类型匹配错误。

◈ 练习 2.2

实现 isSorted 方法，检测 Array[A] 是否按照给定的比较函数排序：

```
def isSorted[A](as: Array[A], ordered: (A,A) => Boolean): Boolean
```

2.5.2 对高阶函数传入匿名函数

在使用高阶函数时，不必非要提供一些有名函数，可以传入*匿名函数*或*函数字面量*。这一点很方便，举例来说，我们可以在 REPL 中用下面的方式测试 findFirst 函数：

```
scala> findFirst(Array(7, 9, 13), (x: Int) => x == 9)
res2: Int = 1
```

这里有一些新的语法，表达式 Array(7,9,13) 是一段“*数组字面量*”，它用 3 个整数构造一个数组。注意构造数组时并没有使用 new 关键字。

语法 (x:Int) => x==9 是一段“*函数字面量*”或“*匿名函数*”。不必先定义一个有名称的方法，可以利用语法的便利，在调用时再定义。这个特定的函数接收一个 Int 类型参数 x，并返回一个 Boolean 类型的值，表示 x 是否等于 9。

通常函数的参数声明在 => 箭头的左边，可以在箭头右边的函数体内使用它们。比如写一个比较两个整数是否相等的函数：

```
scala> (x: Int, y: Int) => x == y
res3: (Int, Int) => Boolean = <function2>
```

在 REPL 结果中的 <function2> 符号表示 res3 是一个接收 2 个参数的函数。如果 Scala 可以从上下文推断输入参数的类型，函数参数可以省略掉类型符号。例如，(x, y) => x < y。我们在下一节会看到一个这样的例子，这本书的后面也会有更多这样的例子。

在 Scala 中函数也是值

当我们定义一个函数字面量的时候，实际上定义了一个包含一个 apply 方法的 Scala 对象。Scala 对这个方法名有特别的规则，一个有 apply 方法的对象可以把它当成方法一样调用。我们定义一个函数字面量 (a, b) => a < b，它其实是一段创建函数对象的语法糖：

```
val lessThan = new Function2[Int, Int, Boolean] {
    def apply(a: Int, b: Int) = a < b
}
```

lessThan 的类型是 Function2[Int, Int, Boolean]，通常写成 (Int, Int) => Boolean。注意 Function2 接口（在 Scala 中是 trait）包含一个 apply 方法，当我们以 lessThan(10,20) 的方式调用函数 lessThan 时它实际是对 apply 方法调用的语法糖：

```
scala> val b = lessThan.apply(10, 20)
b: Boolean = true
```

Function2 只是一个由 Scala 标准库（API 文档：http://mng.bz/qFMr）提供的普通的特质（接口），代表接收两个参数的函数对象。同样在标准库里还提供了 Function1、Function3 等其他函数对象，接收的参数个数从名称里能看出来。因为这些函数在 Scala 中就是普通对象，所以它们也是一等值。通常我们说“函数”这个名词时是指一等函数对象还是一个方法，取决于上下文。

2.6 通过类型来实现多态

或许在前面写 isSorted 函数时你已注意到实现一个多态函数时，各种可能的实现方式明显减少了。针对某种类型 A 的多态函数，唯一可以对 A 进行操作的方式是传入一个函数参数（或按照给出的操作来定义一个函数）。[7] 在某些例子里你会发现对一个多态类型的实现可能被限制为只有一种实现方式。

我们看一个例子，这个函数签名表示它只有一种实现方式。它是执行“部分应用”的高阶函数。函数 partial1 接收一个值和一个带有两个参数的函数，并返回一个带有一个参数的函数。*部分应用*（partial application）这个名词，表示函数被应用的参数不是它所需要的完整的参数：

```
def partial1[A,B,C](a: A, f: (A,B) => C): B => C
```

函数 partial1 有三个类型参数：A、B 和 C。它带有两个参数，参数 f 本身是一个有两个类型分别为 A 和 B 的参数、返回值为 C 的函数。函数 partial1 的返回值也是一个函数，类型为 B=>C。

我们如何继续实现这个高阶函数？结果是能编译通过且符合类型签名逻辑的只有一种实现方式。它看上去就像一个有趣的小逻辑谜题。[8]

看一下返回的类型，partial1 返回值类型是 B=>C，可以写一个接收 B 参数类型的函数字面量：

```
def partial1[A,B,C](a: A, f: (A,B) => C): B => C =
  (b: B) => ???
```

如果你是第一次写匿名函数可能觉得很怪异，B 是从哪儿来的？其实我们只是写了一个“返回一个函数，这个函数接收一个类型为 B 的参数值 b”。在右箭头符号（=>）的右

---

7 技术上 Scala 中所有的值都可以用 == 比较相等性，都可以用 toString 来转成字符串，用 hashCode 得到整型哈希值，但这是从 Java 中继承的缺点。

8 尽管这是一个有趣的谜题，它不是一个纯粹的学术练习。函数式编程在实践中行之有效的唯一方法是引入一堆合适的积木（构件）。这个练习的目的是使用高阶函数，使用 Scala 类型系统引导你编程。

手边（使用问号比较的地方）跟着一个匿名函数的方法体，匿名函数方法体中可以引用值 b，同样可以引用 partial1 方法体中的值 a。[9]

让我们继续，现在来请求类型 B 的值，我们希望让匿名函数返回什么类型？匿名函数的类型签名表明它是类型 C，只有一种方式可以实现。按照签名，函数 f 的返回值正好是 C，所以唯一能得到 C 的方式是传递 A 和 B 的值给 f。也就是：

```
def partial1[A,B,C](a: A, f: (A,B) => C): B => C =
  (b: B) => f(a, b)
```

完成！结果是一个高阶函数接收一个带有两个参数的函数，进行部分应用。即我们有一个 A 和一个需要 A 和 B 产生 C 的函数，可以得到一个只需要 B 就可以产生 C 的函数（因为我们已经有 A 了）。就像我拿一个胡萝卜换你一个苹果和香蕉，你已经给了我一个苹果，只用再给我一个香蕉就可以换胡萝卜了。

◈ **练习 2.3**

我们看另一个柯里化[10]（currying）的例子，把带有两个参数的函数 f 转换为只有一个参数的部分应用函数 f。这里只有实现可编译通过。

```
def curry[A,B,C](f: (A, B) => C): A => (B => C)
```

◇ **练习 2.4**

实现反柯里化（uncurry），与柯里化正相反。注意，因为右箭头 => 是右结合的，A => (B => C) 可以写为 A => B => C。

```
def uncurry[A,B,C](f: A => B => C): (A, B) => C
```

看最后一个例子——函数组合，也就是把一个函数的输出结果当作输入提供给另一个函数。实现这个函数完全取决于它的类型签名。

◈ **练习 2.5**

实现一个高阶函数，可以组合两个函数为一个函数。

```
def compose[A,B,C](f: B => C, g: A => B): A => C
```

这是一个常见的行为，所以 Scala 标准库中的 Function1（带有一个参数的函数接口）提供了 Compose 方法。要对函数 f 和 g 进行组合，只需要简单地写成 f compose g。[11] 同时还提供了一个 andThan 方法，f andThan g 等价于 g compose f：

---

9 在这个内部函数体里，外部的 a 依然在可见范围内。有时候我们说内部函数隐藏了它包含 a 的上下文的环境。

10 这个名字源自发现了这一原则的数学家 Haskell Curry，早先也被 Moses Schoenfinkel 独立发现，但 *Schoenfinkelization* 这个名词没有流行起来。

11 使用库函数来解决这个 compose 练习属于作弊。

```
scala> val f = (x: Double) => math.Pi / 2 - x
f: Double => Double = <function1>

scala> val cos = f andThen math.sin
cos: Double => Double = <function1>
```

对于像这样小的一行层序，还不算困难，但对于真实世界中的大型代码呢？在函数式编程中，被证明是一样的。例如，compose 这样的高阶函数不关心它们所操作的函数是一个上万行代码的巨型函数还是只有一行的简单函数。多态高阶函数的适用范围极其广泛，因为它们不是面向特定领域，而是对发生在很多上下文里的通用模式的抽象。正因为这样，编写大型程序与编写小型程序时的“味道”非常相似。本书将会写很多这种广泛适用的函数。本章给出的练习尝试说服你在写这一类函数时应该采取这种风格。

2.7 小结

本章学习了让我们轻松上手的 Scala 语言基础，以及一些函数式编程概念。我们学习了如何定义一个简单的函数和程序，包括如何使用递归来表达循环。然后引入了一些高阶的函数式思想，做了一些多态函数的练习。我们看到了普通的多态函数实现所存在的严重限制，可以借助于类型来改进实现，这也会在后续章节多次出现。

尽管我们没有写一些大的或复杂的程序，这里讨论的原则是可以延伸的，不管程序大小都适用。接下来我们将使用纯函数来操作数据。

3 函数式数据结构

我们在引言里提到函数式编程不会更新变量，或修改可变的数据结构。这会导致一个紧迫的问题：在函数式编程中可以用哪种数据结构？怎么在 Scala 中定义和操作它们？在这一章，我们将学习并运用函数式数据结构的概念，借用这个机会介绍在函数式编程中如何定义数据类型，了解与之相关的“模式匹配”，并练习编写或泛化一些纯函数。

本章有大量的练习，有些练习可能有一定的挑战，可以在 GitHub（https://github.com/fpinscala/fpinscala）上请求提示或答案，有必要的话可以在线寻求帮助。

3.1 定义函数式数据结构

函数式数据结构，只能被纯函数操作（别惊讶）。记住，纯函数一定不能修改原始数据或产生副作用。因此，函数式数据结构被定义为不可变的。比如，空列表（在 Scala 中写为 `List()` 或 `Nil`）如同整数 3 或 4 一样是永恒不变的。执行 3+4 会产生一个新的数值 7 而不会修改 3 或 4，连接两个 list（对 a 和 b 两个 list 可以写为 `a ++ b`）产生一个新的 list，对输入的两个 list 不做改变。

这是否意味着我们要对数据做很多额外的复制？或许会让你吃惊，答案是否定的。我们后面再谈为什么会这样，但首先让我们先检验一下或许是最普遍存在的函数式数据结构：单向链表。这里定义在本质上与 Scala 标准库中的 List 数据类型相同，这段代码所引入的新语法和概念我们会通过细节来讨论。

示例 3.1 单向链表

```
package fpinscala.datastructures

sealed trait List[+A] ←——List是一个泛型的数据类型，类型参数用A表示。
case object Nil extends List[Nothing] ←——用于表现空List的List数据构造器。
case class Cons[+A](head: A, tail: List[A]) extends List[A] ←——
                                    另一个数据构造器，呈现非空List。注意尾部是另一个
                                    List[A]，当然这个尾部也可能为Nil或另一个Cons。

object List { ←——List伴生对象。包含创建List和对List操作的一些函数。
```

```
    def sum(ints: List[Int]): Int = ints match {
      case Nil => 0
      case Cons(x,xs) => x + sum(xs)
    }
    def product(ds: List[Double]): Double = ds match {
      case Nil => 1.0
      case Cons(0.0, _) => 0.0
      case Cons(x,xs) => x * product(xs)
    }

    def apply[A](as: A*): List[A] =
      if (as.isEmpty) Nil
      else Cons(as.head, apply(as.tail: _*))
}
```

利用模式匹配对一个整数型List进行合计。

空列表的累加值为0。

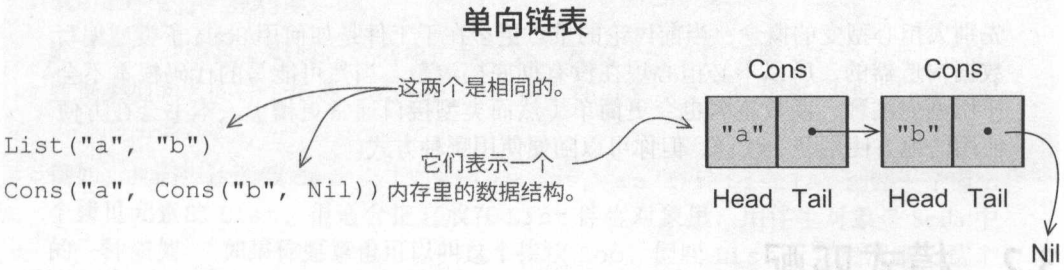
对一个头部是x的列表进行累加，这个过程是用x加上该列表剩余部分的累加值。

可变参数（译注：可以是一个或多个该类型的参数）函数语法。

先看一下数据类型的定义，以 `sealed trait` 关键字开头。通常我们使用 `trait` 关键字引入一种数据类型，`trait` 是一种可以包含一些具体方法（可选）的抽象接口。这里我们定义了一个没有任何方法的 `List` 特质，前边的 `sealed` 关键字意味着这个特质的所有实现都必须定义在这个文件[1]里。

`List` 有两种实现，或者说构造器（每种都由 `case` 关键字引入），表示 `List` 有两种可能的形式。如顶部所示，`List` 如果为空，使用数据构造器 `Nil` 表示，如果非空，使用数据构造器 `Cons`（Construct 的缩写）表示。一个非空 List 由初始元素 `head` 和后续紧跟的也是 List 结构（可能为空）的 `tail` 组成。

```
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```



正如函数可以是多态的，数据类型也可以。通过在 `sealed trait List` 之后添加类型参数 `[+A]`，然后在 `Cons` 数据构造器内部使用 `A`，我们定义 `List` 数据类型所包含的元素类型是多态的，也就意味着对于 `Int` 元素（用 `List[Int]` 表示）、`Double` 元素（用 `List[Double]` 表示）、`String` 元素（用 `List[String]` 表示）等，可以用同样的定义（类型参数 `A` 前面的 `+` 表示 `A` 是发生协变的——更多信息参考注释里有关协变的更多内容）。

1 这里也可以用抽象类代替 trait，它们两个的区别对我们当前的目的并不重要。更多区别看 Scala 语言规范 5.3 节 (http://mng.bz/ R75t)。

一个数据构造器声明，给我们一个函数来构造那种形式的数据类型。下面是一个例子：

```
val ex1: List[Double] = Nil
val ex2: List[Int] = Cons(1, Nil)
val ex3: List[String] = Cons("a", Cons("b", Nil))
```

case object Nil 让我们用 Nil 来构造一个空 List，case class Cons 让我们用 Cons(1,Nil)、Cons("a", Cons("b", Nil)) 等方式来构造任意长度的单向链表。[2] 因为 List 是多态函数，参数化的类型 A 可以实例化为不同的类型。这里 ex2 例子中将类型参数 A 实例化为 Int，而 ex3 例子中实例化为 String。ex1 例子很有趣，Nil 可以是一个 List[Double] 类型的实例。之所以允许这样，因为空 List 中无元素，可以认为是任何类型的 List。

每一种数据构造器同时也引入一种可用于模式匹配的模式，如同函数里的 sum 和 product，接下来会通过更多细节来检验模式匹配。

关于型变（variance）

在 trait List[+A] 声明里，类型参数 A 前边的 + 是一个型变的符号，标志着 A 是协变的或正向（positive）的参数。意味着假设 Dog 是 Animal 的子类，那么 List[Dog] 是 List[Animal] 的子类。（多数情况下，所有类型 X 和 Y，如果 X 是 Y 的子类型，那么 List[X] 是 List[Y] 的子类型）。我们可以在 A 的前面去掉 + 号，那样会标记 List 的参数类型是非型变的（invariant）。

但注意 Nil 继承 List[Nothing]，Nothing 是所有类型的子类型，也就是说使用型变符号后 Nil 可以当成是 List[Int] 或 List[Double] 等任何 List 的具体类型。

先别太担心型变的概念，当前讨论的重点更多在于工件是如何用 Scala 子类型编写数据构造器的，所以不必担心现在没有彻底搞清楚。当然可能写的代码根本不会使用型变注释，函数签名也会更简单（然而类型接口通常更糟）。本书会在方便使用的地方使用型变注释，但你可以随便使用哪种方式。

3.2 模式匹配

让我们看一下函数 sum 和 product 的细节，这两个函数放在 List 单例（object）里，有时也称为 List 的伴生对象（见注释），两个方法都使用了模式匹配：

2 Scala 对任何 case class 或 case object 生成默认的 def toString: String 方法。这对 debug 很方便。你可以看默认的 toString 实现。如果你尝试在 REPL 下用 List 值使用 toString 渲染每一个表达式的结果，例如，Cons(1, Nil) 将打印字符串 "Cons(1, Nil)"，但注意生成的 toString 方法原生是递归的，如果要打印的 List 很长，可能会造成栈溢出。所以你可能想提供一个不同的实现。

```
def sum(ints: List[Int]): Int = ints match {
    case Nil => 0
    case Cons(x,xs) => x + sum(xs)
}

def product(ds: List[Double]): Double = ds match {
    case Nil => 1.0
    case Cons(0.0, _) => 0.0
    case Cons(x,xs) => x * product(xs)
}
```

正如你所期待的那样，sum 函数对空 list 声明的和为 0，非空 list 是第一个元素 x 加上其余元素 xs。[3] 同样 product 函数是对空 list 声明为 1.0，任何以 0.0 作为第一个元素的非空 list 乘积为 0.0，其他的非空 list 是第一个元素乘以其他元素的积。注意这里是递归定义的，这在写操作递归数据类型的函数时很常见，比如 List，在它的 Cons 数据构造器中递归引用自身类型。

模式匹配有点像一个别致的 switch 声明，它可以侵入到表达式的数据结构内部，对这个结构进行检验和提取子表达式。它由一个表达式引入，例如 ds（目标或被检验者）后边跟着一个关键字 match 和一个用花括号封装起来的一系列 case 语句。每一条 case 语句由 => 箭头左边的模式（如 Cons(x,xs)）和 => 箭头右边的结果（如 x * product(xs)）组成。如果目标匹配其中的一种模式，它的结果就是整个 match 表达式的结果。如果目标与多个模式都匹配，Scala 选择第一个匹配的。

Scala 中的伴生对象

除了经常声明数据类型和数据构造器之外，我们也经常声明伴生对象。它只是与数据类型同名的一个单例（object），通常在里面定义一些用于创建或处理数据类型的便捷方法。

例如，我们想让函数 def fill[A](n: Int, a: A): List[A] 创建一个有 n 个拷贝元素的 List，很适合把它放在 List 伴生对象里，用伴生对象是 Scala 中的一种惯例。[4] 如果你愿意也可以叫这个模块 Foo，但叫 List 让它更清晰，这个模块所包含的函数都是与 list 相关的。

让我们再看几个模式匹配的例子：

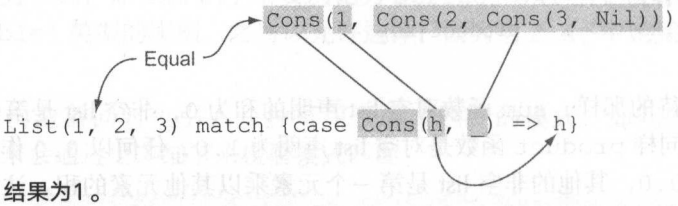
---

3 我们可以对 x 和 xs 用任何名称，但通常是使用 xs、ys、as 或 bs 作为某种序列的变量名，用 x、y、z、a 或 b 作为这个序列的单个元素名。另一个常见的命名习惯是用 h 表示列表的第一个元素（head 的缩写），用 t 表示剩余元素（tail 的缩写），用 l 表示整个列表（list 的缩写）。

4 伴生对象在语言里有一些特定的支持，但与我们的目的不相关。

- `List(1,2,3) match { case _ => 42 }` 结果是 42，这里使用变量模式 `_`，匹配任何表达式。我们可以用 x 或 foo 代替 `_`，但通常使用 `_` 指示一个在匹配结果里不关心的变量。[5]
- `List(1,2,3) match { case Cons(h,_) => h }` 结果是 1，这里使用了一个数据构造器模式结合一个变量模式捕获目标的子表达式。

匹配一个List



结果为1。

- `List(1,2,3) match { case Cons(_,t) => t }` 结果是 `List(2,3)`。
- `List(1,2,3) match { case Nil => 42 }` 结果是运行时匹配错误（MatchError），表示没有表达式与目标匹配。

表达式是否符合模式匹配是由什么决定的？一个模式或许包含只能匹配常量的诸如 3 或 "hi" 这样的字面值；或能匹配任何表达式的以小写字母开头的如 x、xs 这样的变量；或以下画线和只能匹配相应形式的值的如 `Cons(x,xs)` 和 `Nil` 这样的数据构造器（`Nil` 模式只能匹配 `Nil` 单例值，`Cons(h,t)` 或 `Cons(x,xs)` 模式匹配 Cons 类型的值）。模式的组成可以任意嵌套——`Cons(x1, Cons(x2,Nil))` 和 `Cons(y1, Cons(y2, Cons(y3, _)))` 都是有效的模式。如果将模式中的变量分配给目标子表达式，使得它在结构上与目标一致，模式与目标就是匹配的。匹配上的话，结果表达式可以访问这些模式中定义的局部变量。

◈ 练习 3.1

下面的匹配表达式结果是什么？

```
val x = List(1,2,3,4,5) match {
    case Cons(x, Cons(2, Cons(4, _))) => x
    case Nil => 42
    case Cons(x, Cons(y, Cons(3, Cons(4, _)))) => x + y
    case Cons(h, t) => h + sum(t)
    case _ => 101
}
```

强烈建议你在 REPL 下体验一下模式匹配，感受一下它是如何运转的。

5 _（下画线）变量模式被特殊对待，为了忽略目标中的多个部分它可能在模式里被多次使用。

Scala 中的可变参函数（variadic function）

List 单例里的 apply 函数是一个可变参函数，它可以接收零个或多个类型为 A 的参数：

```
def apply[A](as: A*): List[A] =
    if (as.isEmpty) Nil
    else Cons(as.head, apply(as.tail: _*))
```

对一个数据类型，在伴生对象中定义一个可变参数的 apply 方法以便构造这个数据类型的实例是一种惯例。通过命名这个 apply 函数并把它放在伴生对象里，我们可以用 List(1,2,3,4) 或 List("hi", "bye") 这样的语法来创建列表。不管多少个值，只要以逗号分开就好（有时也称之为 List 字面量或字面量语法）。

可变参数函数只是对显式传入一系列（对应 Scala 中的 Seq 类型）元素做了一点语法糖，Seq 是 Scala 集合库中的一个接口，它的实现有 List、Queue、Vector 等类似序列（sequence-like）的数据结构。在 apply 内部 as 参数将被绑定到一个 Seq[A] 类型上，Seq 数据类型包含 head 函数（返回第一个元素）和 tail 函数（返回除了第一个元素之外的其余元素）。特殊的 _* 类型注释允许我们传入一个 Seq 到一个可变参数方法。

3.3 函数式数据结构中的数据共享

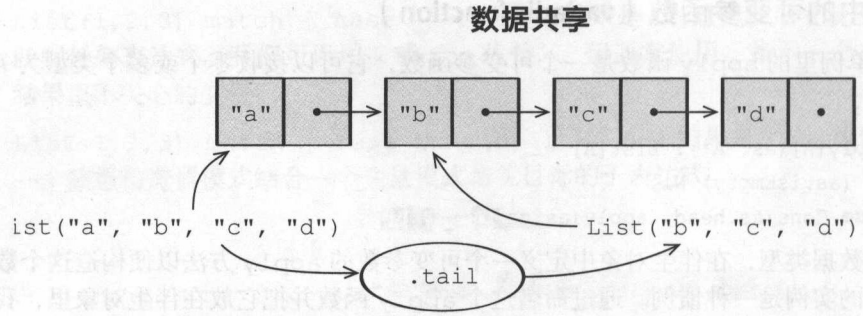
当数据不可变时，我们该怎么写一些例如从 list 中删除元素之类的函数？答案很简单，当我们对一个已存在的列表 xs 在前面添加一个元素 1 的时候，返回一个新的列表，即 Cons(1, xs)。既然列表是不可变的，我们不需要真的去复制一份 xs，可以直接复用它，这也称为*数据共享*。共享不可变数据可以让函数实现更高的效率。我们可以返回不可变数据结构而不用担心后续代码修改它，不需要悲观地复制一份数据以避免对其修改或污染。[6]

以同样的方式，删除一个列表的第一个元素，比如 myList = Cons(x, xs)，只需要返回尾部的 xs。并没有真的删除元素，原始列表 myList 依然可用，丝毫未受影响。我们说函数式数据结构是持久的，意味着已存在的引用不会因数据结构的操作而改变。

让我们试着实现一个有点不一样的函数，以另一种方式来修改列表。可以把这个函数和其他我们写过的函数都放在 List 伴生对象里。

---

6 悲观的复制方式在大型程序里会造成问题。如将可变数据传递给一连串的松散耦合的组件，每个组件都会复制一份自己的数据，因为其他组件可能会修改它。不可变数据在共享时总是安全的，所以我们不必复制数据。我们发现在大型程序里，在实现相同的效果时函数式编程往往比依赖副作用能取得更好的性能，因为函数式能更好地利用数据共享和数据运算。



两个列表在内存中共享相同的数据。.tail方法不会修改原始的列表，它简单地引用原始列表的尾部。不需要防御型的拷贝，因为列表是不可变的

◈ **练习 3.2**

实现 tail 函数，删除一个 List 的第一个元素。注意这个函数的时间开销是常量级的。如果列表是 Nil，在实现的时候会有什么不同的选择？我们在下一章再回顾这个问题。

◈ **练习 3.3**

使用相同的思路，实现函数 setHead 用一个不同的值替代列表中的第一个元素。

3.3.1 数据共享的效率

数据共享通常可以让实现有更好的性能，让我们看几个例子。

◈ **练习 3.4**

把 tail 泛化为 drop 函数，用于从列表中删除前 *n* 个元素。注意，这个函数的时间开销只需要与 drop 的元素个数成正比——不需要复制整个列表。

```
def drop[A](l: List[A], n: Int): List[A]
```

◈ **练习 3.5**

实现 dropWhile 函数，删除列表中前缀全部符合判定的元素。

```
def dropWhile[A](l: List[A], f: A => Boolean): List[A]
```

下面看一个更令人惊讶的例子，利用数据共享的特性将一个列表的所有元素加到另一个列表的后面：

```
def append[A](a1: List[A], a2: List[A]): List[A] =
  a1 match {
    case Nil => a2
    case Cons(h,t) => Cons(h, append(t, a2))
  }
```

注意这个定义只做数据复制，直到第一个列表中没有元素可用，所以它的时间和内存开销只取决于 a1 的长度。如果我们用两个数组实现相同的函数，被迫要复制两个数组中的所有元素到一个结果里，这种情况下不可变链表比数组更有效率。

◈ **练习 3.6**

不是所有的实现都这么令人满意，实现一个 init 函数，返回一个列表，它包含原列表中除了最后一个元素之外的所有元素。比如，传入 List(1,2,3,4) 给 init 函数会返回 List(1,2,3)，为什么这个函数不能实现同 tail 一样的常量级时间开销？

```
def init[A](l: List[A]): List[A]
```

因为单向链表的结构，任何时间我们想要替代 Cons 的尾部，甚至是链表的最后一个 Cons，我们必须复制所有之前的 Cons 对象。所有关于更高效支持不同操作方式的纯函数式数据结构，其实都是找到一种聪明的方式来利用数据共享。我们不会在这里涉及这种数据结构，当前我们乐于使用现有的函数式数据结构而非再写一些新的。举一个例子，在 Scala 标准库中有一个纯函数式序列的实现：Vector（文档：http:// mng.bz/Xhl8），对它的随机访问、更新、head、tail、init，以及在序列前后添加元素操作都是常量级时间开销。可以通过本章的注释和链接阅读更多如何设计这种数据结构的内容。

3.3.2 改进高阶函数的类型推导

高阶函数如 dropwhile 经常传递匿名函数。看一个典型的例子，回忆一下 dropwhile 的签名：

```
def dropWhile[A](l: List[A], f: A => Boolean): List[A]
```

当我们传入一个匿名函数 f 来调用它时，我们必须指定它的参数类型，这里是 x：

```
val xs: List[Int] = List(1,2,3,4,5)
val ex1 = dropWhile(xs, (x: Int) => x < 4)
```

ex1 的值是 List(4,5)。

不幸的是我们需要声明 x 类型是 Int。dropwhile 的第一个参数是一个 List[Int]，所以函数的第二个参数必须接收 Int 类型。Scala 可以推导这种情况，如果我们把 dropwhile 的参数分成两组：

```
def dropWhile[A](as: List[A])(f: A => Boolean): List[A] =
    as match {
        case Cons(h,t) if f(h) => dropWhile(t)(f)
        case _ => as
    }
```

这个版本的 dropwhile 的语法看起来像 dropwhile(xs)(f)，这里 dropwhile(xs) 返回一个函数，然后对这个函数传入参数调用（换句话说 dropwhile 是柯

里化的[7]）。使用这种方式把参数分组的主要目的是帮助类型推导。现在我们可以在使用 dropwhile 时不加类型标注。

```
val xs: List[Int] = List(1,2,3,4,5)
val ex1 = dropWhile(xs)(x => x < 4)
```

注意，x 变量没有使用类型标注。

一般来讲，当函数定义包含多个参数组时，参数组里的类型信息从左到右传递。这里第一个参数组确定 A 类型参数为 Int，所以 x => x < 4 里的类型标注可以不需要。[8]

我们常通过将函数参数分组排序成多个参数列表，来最大化地利用类型推导。

3.4 基于 list 的递归并泛化为高阶函数

再看一下 sum 和 product 的实现。我们已经稍微简化了 product 的实现。不引入检测 0.0 的短路逻辑：

```
def sum(ints: List[Int]): Int = ints match {
    case Nil => 0
    case Cons(x,xs) => x + sum(xs)
}

def product(ds: List[Double]): Double = ds match {
    case Nil => 1.0
    case Cons(x, xs) => x * product(xs)
}
```

注意这两个定义很相似，它们操作不同的类型（List[Int] 与 List[Double]），除此之外的差异还有在 List 为空时的返回值（sum 返回 0，product 返回 1.0），以及对结果的组合操作（sum 是 +，product 里是 *）。如果再遇到类似重复的情况，可以把子表达式放到函数参数里来进行泛化。如果一个子表达式引用任何局部变量（+ 操作引用了模式中的 x 和 xs 局部变量，product 里也存在相同的情况），把子表达式放入一个接收这些变量作为参数的函数。现在就改一下，函数的参数包含当列表为空时返回的值，以及列表非空时用于将元素添加到结果的函数。[9]

示例 3.2 右折叠的简单运用

```
def foldRight[A,B](as: List[A], z: B)(f: (A, B) => B): B =
    as match {
        case Nil => z
```

再次，把f参数独立出来，放在as和z参数后边，是为了让类型系统能推导出f的输入类型。

7　回忆一下前一章，有两个参数的函数可以表现为：只接收一个参数，返回另一个函数，这个返回的函数用另一个参数做它的参数。

8　不幸的是 Scala 编译器有一个限制，其他函数式语言比如 Haskell 和 OCaml 则能提供完整的类型推导，对这两种编程语言而言类型注释几乎不需要。进一步阅读更多信息请看本章的笔记和链接。

9　在 Scala 标准库中，foldRight 是 List 中的一个方法，它的参数是柯里化的，为了更好地进行类型推导。

```
        case Cons(x, xs) => f(x, foldRight(xs, z)(f))
    }

def sum2(ns: List[Int]) =
    foldRight(ns, 0)((x,y) => x + y)

def product2(ns: List[Double]) =
    foldRight(ns, 1.0)(_ * _)  <——_ * _是对(x,y) => x*y更简练的写法（参考下面的注释）。
```

foldRight 函数不是面向特定的一种元素类型，泛化的返回值不必一定与 list 中的元素类型相同！一种描述 foldRight 是做什么的方式是用 z 和 f 替换列表 Nil 和 Cons 的构造器。举例来说：

```
Cons(1, Cons(2, Nil))
f   (1, f   (2, z  ))
```

来看一个完整的例子。我们将通过不断代入 foldRight 所使用的定义，跟踪一下 foldRight(Cons(1, Cons(2, Cons(3, Nil))), 0)((x,y) => x + y) 的运算过程。像这样的程序跟踪将贯穿整本书。

```
foldRight(Cons(1, Cons(2, Cons(3, Nil))), 0)((x,y) => x + y)
1 + foldRight(Cons(2, Cons(3, Nil)), 0)((x,y) => x + y)  <——将foldRight替换为它的
1 + (2 + foldRight(Cons(3, Nil), 0)((x,y) => x + y))          定义。
1 + (2 + (3 + (foldRight(Nil, 0)((x,y) => x + y))))
1 + (2 + (3 + (0)))
6
```

注意 foldRight 在开始迭代之前必须一路遍历到列表的末尾（在处理过程中不断压栈）。

针对匿名函数的下画线

匿名函数 (x, y) => x + y，在 x 和 y 类型可以被 Scala 推导的情况下可以缩写为 _ + _。对那些函数参数只在函数体中出现一次的函数，这种缩写很适用。匿名函数表达式中的每一个下画线，例如 _ + _ 会引入一个新的（未命名）函数参数，并对其引用。参数的引入按照从左到右的顺序，下面是更多的例子：

```
_ + _  <——(x,y) => x + y
_ * 2  <——x => x * 2
_.head <——xs => xs.head
_ drop _ <——(xs,n) => xs.drop(n)
```

审慎地使用这种语法。这种语法在表达式如 foo(_, g(List(_ + 1), _)) 中会导致不清晰，在 Scala 语言规范里对基于下画线的匿名函数的范围有严格的规定。推荐使用常规的有名函数参数。

◈ 练习 3.7

在入参是 0.0 时用 foldRight 实现 product 是否可以立即停止递归并返回 0.0？为什么可以或者为什么不可以？想想看如果你对一个大列表调用 foldRight 会有多少短路发生。这个问题有点深，我们在第 5 章再来回顾。

◈ **练习 3.8**

当你对 `foldRight` 传入 `Nil` 和 `Cons` 时，看看会发生什么？例如：`foldRight(List(1,2,3,4), Nil:List[Int])(Cons(_,_))`。[10] 说到 `foldRight` 和 `List` 数据结构之间的关系，你有什么想法？

◈ **练习 3.9**

使用 `foldRight` 计算 List 的长度。

```
def length[A](as: List[A]): Int
```

◈ **练习 3.10**

我们实现的 `foldRight` 不是尾递归，如果 List 很大可能会发生 `StackOverflowError`（我们称之为非栈安全的）。说服自己接收这种情况，然后用尾递归方式写另一个通用的列递归函数 `foldLeft`。签名如下：[11]

```
def foldLeft[A,B](as: List[A], z: B)(f: (B, A) => B): B
```

◈ **练习 3.11**

写一下 `sum`、`product` 函数，和一个用 `foldLeft` 计算列表长度的函数。

◈ **练习 3.12**

写一个对原列表元素颠倒顺序的函数（`List(1,2,3)` 返回 `List(3,2,1)`），看看是否可用一种折叠（fold）实现。

◇ **练习 3.13**

难：你是否能根据 `foldRight` 来写一个 `foldLeft`？有没有其他变通方式？通过 `foldLeft` 来实现 `foldRight` 很有用，因为这会让我们以尾递归的方式实现。意味着不管列表有多大都不会产生栈溢出。

◈ **练习 3.14**

根据 `foldLeft` 或 `foldRight` 实现 `append` 函数。

◈ **练习 3.15**

难：写一个函数将一组列表连接成一个单个列表。它的运行效率应该随所有列表的总长度线性增长。试着用我们已经定义过的函数。

---

10 类型注释 `Nil::List[Int]` 在这里是需要的，否则 Scala 将推断 `foldRight` 中的 `B` 类型参数为 `List[Nothing]`。

11 再次提醒，`foldLeft` 是 Scala 标准库中的 `List` 的一个方法，它也同样是柯里化的，为了更好地进行类型推导。你可以写为 `mylist.foldLeft(0.0)(_ + _)`。

3.4.1 更多与列表相关的函数

还有很多与列表相关的很有用的函数，我们再介绍几个，一些针对泛化函数的额外的练习帮你熟悉处理列表的通用模式。在这一节结束后，你不会对什么时候使用这些函数没有意识。检查是否存在可能的方式对处理列表的显式递归函数进行泛化，是一个好习惯。这样的话，你会形成自己的直觉，用来发现（重新发现）这些函数。

◈ **练习 3.16**

写一个函数，用来转换一个整数列表，对每个元素加 1（记住这应该是一个纯函数，返回一个新列表）。

◈ **练习 3.17**

写一个函数，将 `List[Double]` 中的每一个值转为 `String`，你可以用表达式 `d.toString` 将 `Double` 类型的值转换为 `String`。

◈ **练习 3.18**

写一个泛化的 `map` 函数，对列表中的每个元素进行修改，并维持列表结构。签名如下：[12]

```
def map[A,B](as: List[A])(f: A => B): List[B]
```

◈ **练习 3.19**

写一个 `filter` 函数，从列表中删除所有不满足断言的元素，并用它删除一个 `List[Int]` 中的所有奇数。

```
def filter[A](as: List[A])(f: A => Boolean): List[A]
```

◈ **练习 3.20**

写一个 `flatMap` 函数，它跟 `map` 函数有些像，除了传入的函数 `f` 返回的是列表而非单个结果。这个 `f` 所返回的列表会被塞到 `flatMap` 最终所返回的列表。签名如下：

```
def flatMap[A,B](as: List[A])(f: A => List[B]): List[B]
```

例如：`flatMap(List(1,2,3))(i=>List(i,i))` 结果是 `List(1,1,2,2,3,3)`。

◈ **练习 3.21**

用 `flatMap` 实现 `filter`。

◈ **练习 3.22**

写一个函数，接收 2 个列表，通过对相应元素的相加构造出一个新的列表。比如，`List(1,2,3)` 和 `List(4,5,6)` 构造出 `List(5,7,9)`。

12 在 Scala 标准库中，map 和 flatMap 都是 List 的方法。

◈ 练习 3.23

对刚才的函数泛化，不只针对整数或相加操作。将这个泛化函数命名为 zipWith。

标准库中的列表

列表（List）存在于 Scala 标准库中，在接下来的章节我们将使用标准库中的版本。我们写的 List 和标准库中的 List 不同的是，Cons 在标准库的版本写为 ::，它使用右结合，[13] 所以 1::2::Nil 等于 1::(2::Nil)，也等于 List(1,2)。进行模式匹配时，case Cons(h, t) 变为了 case h::t，在你写 case h::h2::t 这种模式匹配以提取更多 List 中的第一个元素时，免去了必须嵌套的小括号。

标准库中的 List 还有一系列有用的方法，在读完 API 文档后可以在 REPL 下体验一下这些方法。它们都是 List[A] 里定义的方法，而不像我们在这一章对这些函数都是独立定义的：

- def take(n: Int): List[A]——返回一个由当前列表中前 n 个元素构成的列表。
- def takeWhile(f:A=>Boolean):List[A]——返回当前列表中最长的前缀列表，这些元素都必须满足断言 f。
- def forall(f: A => Boolean): Boolean——如果所有元素都满足断言 f 返回 true。
- def exists(f: A => Boolean): Boolean——如果存在任何一个元素满足断言 f 返回 true。
- scanLeft 和 scanRight——就像 foldLeft 和 foldRight，但它们都返回部分结果而非最终的累计值列表。

推荐你在本章学习结束后浏览一下 Scala API 文档，看看里面还有什么函数。如果你发现自己正在写一个对列表操作的显式递归函数，检查一下标准库 API 中是否已经存在。

3.4.2 用简单组件组合 list 函数时的效率损失

List 的一个问题是，虽然我们可以按照非常通用的函数来表达操作和算法，但实现的结果往往不太高效。我们最终可能会对相同的输入进行多次传递，或者不得不写一些显式的递归循环，运行提前终止。

◈ 练习 3.24

难：实现 hasSubsequence 方法，检测一个 List 子序列是否包含另一个 List，比如 List(1,2,3,4) 包含的子序列有 List(1,2)、List(2,3) 和 List(4) 等。或

13 在 Scala 中，所有以 : 命名结尾的方法都是右关联的，也就是说，表达式 x::xs 实际是方法调用：xs.::(x)，反过来调用数据构造器 ::(x, xs)。更多信息请看 Scala 语言规范。

许找到一种简洁高效的纯函数式实现有些困难，没关系，先用最自然的想法实现，我们在第 5 章之后再回顾，到时再改进这个实现。注意：任意两个值 x 和 y 在 Scala 中可以使用表达式 x==y 来比较它们是否相等。

```
def hasSubsequence[A](sup: List[A], sub: List[A]): Boolean
```

3.5 树

List 只是我们所说的*代数数据类型*（ADT）的例子之一（这里 ADT 有点容易混淆，它有时在其他场合表示*抽象数据类型*）。ADT 是由一个或多个数据构造器（data constructor）所定义的数据类型，每个构造器可以包含零个或多个参数。数据类型（data type）是其数据构造器（data constructor）的累加（sum）或联合（union），每个数据构造器又是它的参数的*乘积*（product），所以称之为*代数数据类型*（algebraic data type）。[14]

Scala 中的元组（Tuple）

Tuple 也是代数数据类型，它们与我们之前写的 ADT 类似，但有特定的语法：

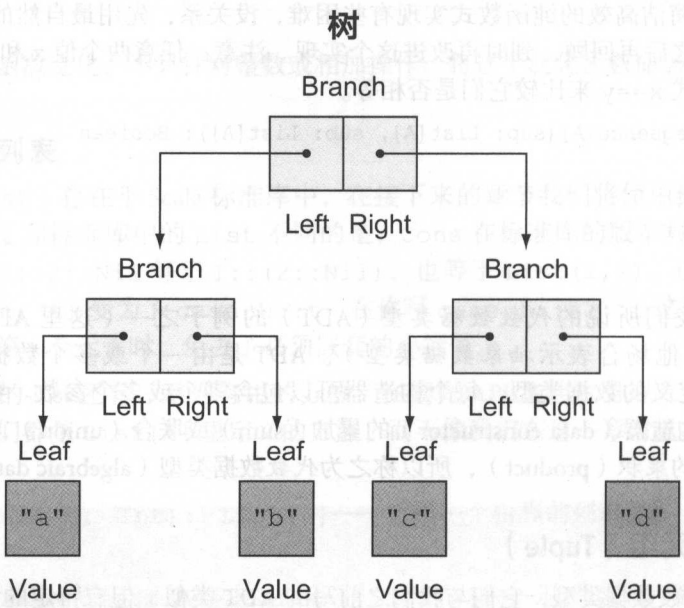
```
scala> val p = ("Bob", 42)
p: (java.lang.String, Int) = (Bob,42)

scala> p._1
res0: java.lang.String = Bob

scala> p._2
res1: Int = 42

scala> p match { case (a,b) => b }
res2: Int = 42
```

这个例子中 ("Bob", 42) 是一个类型为 (String, Int) 的数据对（pair），它其实是 Tuple2[String, Int] 的语法糖（API：http://mng.bz/1F2N）。我们可以提取这个数据的第一个或第二个元素（Tuple3 还可以通过 _3 方法提取第三个元素，以此类推），还可以对这个数据对进行模式匹配，就像其他 case 类那样。更多元数的元组也类似。有兴趣可以在 REPL 下体验一下。

代数数据类型被用于定义其他数据结构，让我们定义一个简单的二叉树数据结构：

```
sealed trait Tree[+A]
case class Leaf[A](value: A) extends Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]
```

14 命名不是巧合，在 ADT 类型与数字的加法和乘法之间更深的关系超出了本书的范围。



模式匹配为我们操作 ADT 中的元素提供了一种便利方式，让我们尝试写一些函数。

◈ **练习 3.25**

写一个 `size` 函数，统计一棵树中的节点数（叶子节点和分支节点）。

◈ **练习 3.26**

写一个 `maximum` 函数，返回 `Tree[Int]` 中的最大的元素（提示，在 Scala 中使用 `x.max(y)` 或 `x max y` 计算整数 `x` 和 `y` 的最大值）。

◈ **练习 3.27**

写一个 `depth` 函数，返回一棵树中从根节点到任何叶子节点最大的路径长度。

◈ **练习 3.28**

写一个 `map` 函数，类似于 `List` 中的同名函数，接收一个函数，对树中每个元素进行修改。

代数数据类型与封装性

或许会有人对代数数据类型提出质疑，认为它公开了一个类型的内部表征，因而违反了数据的封装性。在函数式编程中，我们对封装的关注方式不同——如果公开暴露的是可变状态，可能会导致 bug 或违反不变性，然而函数式编程中通常不

会有这种微妙的可变状态。暴露一个类型的数据构造器没有什么问题，这样做的决定很像对数据类型的公共 API 应该怎么暴露时的决定。[15]

使用 ADT 的典型场景是一组封闭的样例（固定的）。对于 `List` 和 `Tree`，改变数据集的构造函数也显然改变了数据类型本身。`List` 本质是一个单向链表，由两个样例 `Nil` 和 `Cons` 组成它可用的公开 API。我们当然可以写一些代码用更抽象的 API 来处理而非用 `List` 来处理（会在后面看到这样的例子），但这种信息隐藏可以用一个独立的层来处理而非直接放入 `List` 中。

◈ **练习 3.29**

泛化 `size`、`maximum`、`depth` 和 `map`，写一个新的函数 `fold`，对它们的相似性抽象。按照更加通用的函数标准来重新实现它们。你能对这个 `fold` 函数与 `List` 中的 left 和 right fold 做一下类比吗？

3.6 小结

本章涵盖了一些重要的概念，介绍了代数数据类型和模式匹配，展示了如何实现纯函数式数据结构，包括一个单向链表。希望通过本章的练习，你可以更轻松地写纯函数并泛化它们。我们后续章节会继续加强这些技能。

15 也可以在 Scala 中暴露这样的模式，例如 `Nil` 和 `Cons` 独立于类型的真实数据构造器。

4 不是用异常来处理错误

我们在第 1 章简单说明过，抛出异常会产生副作用。不过代码中若不能抛出异常，又该用什么替代呢？这一章我们将学习以函数式方法抛出和处理错误的基本原则。最重要的是我们可以用普通的值表现失败和异常，可以通过高阶函数抽象出错误处理和恢复的常用模式。在函数式解决方案中，以值的方式返回错误更安全，符合引用透明，并且可以通过高阶函数保存异常的优点——统一错误处理逻辑。通过本章课程进一步观察异常和讨论它们的一些问题，将看到函数式的方式是如何工作的。

与我们在之前章节中创建我们自己的 `List` 数据类型的理由相同，这一章我们也将重新创建两个 Scala 标准库中的类型：`Option` 和 `Either`。主要目的是为了加强你对这些类型如何用于错误处理的理解。在完成本章课程之后，应该可以自如地使用 Scala 标准库版本的 `Option` 和 `Either`（虽然你会注意到标准库版本的这两个类型缺乏在本章出现的一些有用的函数）。

4.1 异常的优点与劣势

为什么异常破坏了引用透明，会有怎样的问题？让我们看一个简单的例子，我们定义一个会抛出异常的函数并调用它。

示例 4.1 抛出和捕获异常

```
def failingFn(i: Int): Int = {
    val y: Int = throw new Exception("fail!")
    try {
        val x = 42 + 5
        x+y
    }
    catch { case e: Exception => 43 }
}
```

val y: Int = ...声明y是一个Int类型并给它赋等号右边的值。

catch块里是一个模式匹配，就像之前所见过的。case e: Exception是一个匹配任何Exception的模式，它把异常的值绑定到变量e上。这个匹配结果返回值43。

在 REPL 下调用 `failingFn` 会给出预期错误：

```
scala> failingFn(12)
java.lang.Exception: fail!
  at .failingFn(<console>:8)
  ...
```

我们可以证明 y 不是引用透明的，回忆一下引用透明的概念，表达式可以被它引用的值替代，这种替代保持程序的含义。如果我们对 x+y 表达式中的 y 替代为 `throw new Exception("fail!")` 会产生不同的结果。因为 try 代码块内部的异常会被捕获并返回 43：

```
def failingFn2(i: Int): Int = {
    try {
        val x = 42 + 5
        x + ((throw new Exception("fail!")): Int)
    }
    catch { case e: Exception => 43 }
}
```

对于一个抛出异常的表达式可以设定为任何类型；这里我们标注为Int类型。

可以在 REPL 下展示：

```
scala> failingFn2(12)
res1: Int = 43
```

理解引用透明的另一种方式是引用透明的表达式不依赖上下文，可以本地（locally）推导，而那些非引用透明的表达式是依赖上下文的，并且需要全局推导。举例来说，引用透明的表达式 42+5 如果嵌入于一个更大的表达式中，并不会对这个更大的表达式产生依赖，它总是等于 47。但 `throw new Exception("fail")` 表达式是取决于上下文的（如我们刚刚示范的）。它可能会呈现不同的含义，取决于它嵌入的 try 代码块。

异常存在的两个主要问题：

- 正如我们所讨论的，*异常破坏了引用透明并引入了上下文依赖*，让替代模型的简单推导无法适用，并可能写出令人困惑的代码。源自于坊间的一个习惯是建议异常应该只用于错误处理而非控制流。
- 异常不是类型安全的。failingFn, `Int => Int` 类型没有告诉我们可能会发生什么样的异常。编译器当然也不会强迫 failingFn 的调用者决定如何处理这些异常。如果我们忘记检测 failingFn 的异常，直到运行时才被检测到。

检测异常

Java 的异常检测最低限度地强制了是处理还是抛出错误，但它们导致了调用者对签名模板的修改。更重要的是它们不适用于高阶函数，因为高阶函数不可能感知由它的参数引起的特定的异常。例如，考虑对 List 定义的 map 函数：

```
def map[A,B](l: List[A])(f: A => B): List[B]
```

这个函数很清晰，高度泛化，与使用检测异常不一样的是——我们不能对每一个被 f 抛出的异常的检测都有一个版本的 map。甚至如果我们想要这样做，map 怎么知道可能会产生什么样的异常？这也是为什么要泛化代码，甚至在 Java 中常常借用运行时异常（RuntimeException）或某些常见检测的异常类型。

我们希望其他选择能刨除异常的这些缺点，但我们又不希望它会失去异常最主要的一个好处：*整合集中的错误处理逻辑*，而非被迫在整个代码库发布这个逻辑。这里用的技术

基于一个早已存在的理念：替代抛出异常，返回一个值来指示异常情况发生。这种理念或许对那些使用过 C 的返回码来处理异常的人很熟悉。但这里不是使用错误码，我们引入一种新的泛型类型来描述这些“可能存在定义的值”，并使用高阶函数来封装这种处理和传播异常的通用模式。不像 C 风格的错误码，我们使用的错误处理策略是完全类型安全的（completely type-safe），并且能得到类型检测的帮助，以最小的语法噪音让我们有效地处理错误。接下来看看它是怎么工作的。

4.2 异常的其他选择

考虑一种现实中可能使用异常的情况，看看有哪些可以替代的方法。这里有一个函数的实现，计算列表的平均值，它没有定义空列表的情况：

```
                    Seq是各种线性序列类集合的公共接口。更多信息
                ┌──参考API文档(http://mng.bz/f4k9)。
                ▼
def mean(xs: Seq[Double]): Double =
    if (xs.isEmpty)
      throw new ArithmeticException("mean of empty list!")
    else xs.sum / xs.length  ←——sum是当Seq的元素都是数字类型时才定义一个方法。标准库
                                 里是使用隐式转换实现这个技巧的，我们先不去深究。
```

mean 函数是一个部分函数（partial function）：它对一些输入没有做定义。如果一个函数对那些非隐式的输入类型做了一些假设，那它是一个典型的部分函数。[1] 这种情况下可以抛出异常，但我们还有其他选择。看一下对 mean 使用这种选择的例子。

第一种可能是返回某个伪造的 Double 类型的值。可以对所有情况简单地返回 xs.sum/xs.length，对于输入为空的情况，0.0/0.0 返回 Double.NaN，或返回其他报警值。在其他情况下，我们可以返回 null 代替需要类型的值。这个用于如何处理错误的普通类常用在那些没有异常的语言里。出于以下几个理由我们拒绝采用这种方案：

- 它允许错误“无声”的传播——调用者可能忘了检测这种情况也不会被编译器警告，可能导致后续的代码不能正常工作。通常代码中的错误要很久才会被发现。
- 除了易错性，使用显示的 if 声明来检测是否调用者收到了“真正”的结果，还导致在调用点产生一堆代码模板（boilerplate）。如果调用多个函数，代码模板将被放大，每个使用错误码的地方都得检查并以某种方式聚合起来返回。
- 它不适用于多态代码。对某些输出类型，甚至没有我们想要的报警值！例如 max 函数，按照自定义的比较函数从序列中找出最大值：def max[A](xs: Seq[A]) (greater: (A,A) => Boolean): A。如果输入为空，无法发明一个 A 类型的值。null 也不能用在这里，因为 null 只对非基础类型有效，而 A 可能是一个基础类型，比如 Double 或 Int。

---

1 如果一个函数对一些输入没有结果，它可能是部分函数。我们在这儿不讨论部分函数的形式，既然它是一个不可恢复的错误，也就不存在最佳的处理方式。看这一章笔记了解更多有关部分函数的内容。

- 它需要一个特定的策略或调用约定——合理使用mean函数需要调用者做一些事情。像这种特定策略的方法，将很难把它传递给高阶函数，因为高阶函数对待所有参数都是一致的。

第二种可能，针对那些输入不知道怎么处理的情况，强迫调用者提供一个参数告诉我们该如何处理。

```
def mean_1(xs: IndexedSeq[Double], onEmpty: Double): Double =
  if (xs.isEmpty) onEmpty
  else xs.sum / xs.length
```

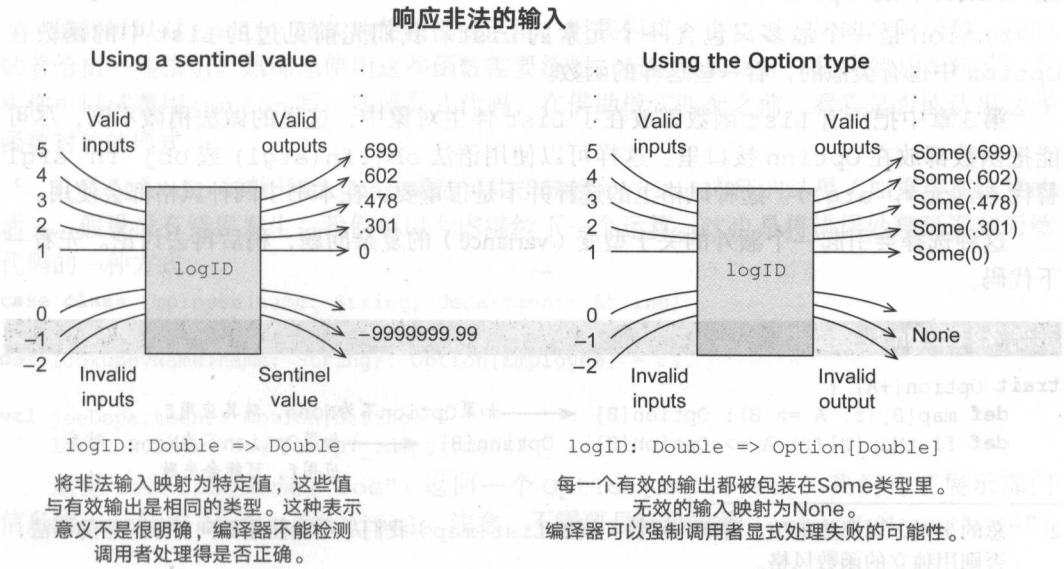
这样让mean成为了一个完全函数（total function），但有一些缺点——它需要直接调用者知道如何处理未定义的情况，限制它们返回一个Double。如果mean作为所调用的一个复杂运算中的一部分，没有定义的话是否终止运算？或者在这个复杂运算里遇到这种情况我们希望选择一种不同的分支？简单传递一个onEmpty参数并没有给我们这种选择的自由。

我们需要一种方式能推迟决定如何处理未定义的情况，可以在最适合的时候去处理。

4.3 Option 数据类型

解决方案是在返回值类型时明确表示该函数并不总是有答案。可以认为这是用于推迟调用者的错误处理策略。我们引入一种新的类型Option。如之前提到过的，这个类型也存在于Scala标准库中，但出于教学目的我们在这里重新创建一个：

```
sealed trait Option[+A]
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```



Option 有两种情况：已被定义的情况对应的是 Some；未被定义的情况对应的是 None。

我们可以用 Option 来定义 mean 函数。比如：

```
def mean(xs: Seq[Double]): Option[Double] =
    if (xs.isEmpty) None
    else Some(xs.sum / xs.length)
```

返回值类型现在反映了一种可能性：结果不一定总被定义。我们依然对函数返回一个声明的类型（现在用 Option[Double]），所以 mean 现在是一个完全函数，对每一个输入的值都有一个对应输出类型的值。

4.3.1 Option 的使用模式

部分函数在编程中大量存在，Option（以及稍后讨论的 Either 数据类型）在函数式编程中常被用于处理部分函数。你会看到 Option 的使用贯穿整个 Scala 标准库，例如：

- 在 Map 中查找一个 key（http:// mng.bz/ha64），返回 Option。
- 在 List 和其他可迭代集合（http:// mng.bz/Pz86）中定义的 headOption 和 lastOption 返回包含第一个或最后一个元素的 Option，如果这个集合不为空的话。

不止这一个例子，我们会看到 Option 出现在很多不同的情况下。它便于我们通过高阶函数提取错误处理的通用模式，让我们避免写一些惯用的异常处理的模板代码。在这一节，会涉及一些基础函数与 Option 来一同工作。我们的目标不在于你马上能畅快地使用这些函数，先熟悉起来，自己控制进度；当你要写一些函数式代码处理错误时可以回过头来再看一下本章的内容。

基础函数中的 Option

Option 是一个最多只包含一个元素的 List。我们先前见过的 List 中的函数在 Option 中也有类似的，看一些这样的函数。

第 3 章中把所有 List 函数都放在了 List 伴生对象中，这里的做法稍微不同，尽可能把函数都放在 Option 接口里。这样可以使用语法 obj.fn(arg1) 或 obj fn arg1 替代 fn(obj, arg1)。这种风格上的选择并不是很重要，在本书中两种风格都会使用。[2]

这种选择会引起一个额外的关于型变（variance）的复杂问题，稍后再去讨论。先看一下代码。

示例 4.2 Option 数据类型

```
trait Option[+A] {
    def map[B](f: A => B): Option[B]          <——如果Option不为None，对其应用f。
    def flatMap[B](f: A => Option[B]): Option[B]   <——如果Option不为None，对其
                                                    应用f，可能会失败。
```

2 总的来说，对那些单独、清晰的函数（例如 List.map）我们尽可能使用面向对象风格的语法，否则用独立的函数风格。

```
  def getOrElse[B >: A](default: => B): B ←——B >: A表示B类型参数必须是A的父类型。
  def orElse[B >: A](ob: => Option[B]): Option[B] ←——不对ob求值，除非需要。
  def filter(f: A => Boolean): Option[A]  ←——如果值不满足f，转换Some为None。
}
```

这里有一些新的语法，在 `getOrElse`（还有 `orElse` 中的相似标注）中的 `default: => B` 类型标注表示参数类型是 `B`，但不是立即求值，而是函数需要时才求值。先别太担心，我们会在下一章讨论更多“非严格求值”的概念。在 `getOrElse` 和 `orElse` 中的 `B >: A` 类型标注表示 `B` 必须等于 `A` 类型或者是 `A` 类型的父类型（supertype）。`Option[+A]` 声明表示 `A` 是协变的（covariant）。本章的注释里有更多详情，有点复杂，但这种复杂却是 Scala 类型系统必需的。幸运的是这里的主要目的并不是要你完全理解子类型和型变。

◈ 练习 4.1

对 `Option` 实现之前的所有函数，在实现每一个函数时试着考虑它有什么意义，在什么场景下使用。接下来会探索何时使用这些函数。这里对解决这个练习给出一些提示：

- 使用模式匹配也可以，虽然可以不借助模式匹配实现这些方法，除了 `map` 和 `getOrElse`。
- 对 `map` 和 `flatMap`，类型签名应该足以确定它的实现。
- `getOrElse` 返回结果是 `Option` 类型，如果 `Option` 为 `None` 返回给定的默认值，否则结果被封装在 `Some` 对象里。
- 如果定义了的话，`orElse` 返回第一个 `Option`；否则返回第二个 `Option`。

基础 Option 函数的使用场景

虽然可以对 `Option` 显式地使用模式匹配，但我们将一直使用上述的高阶函数。我们试着给出一些指引。熟练地使用这些函数需要练习，但这里的目标只是达到基本熟悉，下次你可以试着用 `Option` 写一些函数式代码，在借助模式匹配之前，看看是否能认出这些函数封装的模式。

让我们从 `map` 函数开始，`map` 函数可用于转换 `Option` 内部的结果（如果存在结果的话）。假设没有错误发生，我们可以考虑继续下一个运算。这也是将错误处理延迟到后续代码的一种方式。

```
case class Employee(name: String, department: String)

def lookupByName(name: String): Option[Employee] = ...

val joeDepartment: Option[String] =
  lookupByName("Joe").map(_.department)
```

这里 `lookupByName("Joe")` 返回一个 `Option[Employee]`，我们为了展示部门信息用 map 转成 `Option[String]`。注意，不需要显式地检测 `lookupByName("Joe")`

的结果；继续下一步运算，就当作在 map 的参数内部没有错误发生。如果 employees-ByName.get("Joe") 返回 None，它会终止后续的运算，这种情况 map 根本不会调用 _.department 函数。

```
                                    ┌──→返回Joe的部门，如果Joe是员工。
lookupByName("Joe").map(_.department)
                                    └──→返回None，如果Joe不是员工。

                                    ┌──→返回Some(manager)，如果Joe是一个经理。
lookupByName("Joe").flatMap(_.manager)
                                    └──→返回None，如果Joe不是员工或不是经理。

                                                    ┌──→返回Joe的部门，如果Joe的部门存在。
lookupByName("Joe").map(_.department).getOrElse("Default Dept.")
                                                    └──→返回“缺省部门”，如果Joe的部门不存在。
```

flatMap 也相似，除非我们提供的转换函数自身失败。

◈ 练习 4.2

根据 flatMap 实现一个 variance（方差）函数。如果一个序列的平均值是 m，variance 是对序列中的每一个元素 x 进行 math.pow(x-m, 2)。维基百科中对方差的定义（http://mng.bz/0Qsr）。

```
def variance(xs: Seq[Double]): Option[Double]
```

如同示例里方差函数的实现，用 flatMap 可以对多个阶段构造成一个运算。每个阶段都可能失败，运算中遇到第一个失败便会马上终止，因为 None.flatMap(f) 会立即返回 None，不会再运行 f。

如果成功的结果不匹配给定的预判，可以用 filter 将成功的结果转换成失败的结果。一个通用的模式是调用 map、flatMap 和（或）filter 来转换 Option，在最后用 getOrElse 来处理错误：

```
val dept: String =
  lookupByName("Joe").
  map(_.dept).
  filter(_ != "Accounting").
  getOrElse("Default Dept")
```

getOrElse 这里被用于将 Option[String] 转换为一个 String，通过提供一个默认的部门信息，以防 "Joe" 不存在于 Map 中或 "Joe" 的部门是 "Accounting"。

orElse 与 getOrElse 相似，不同之处在于如果第一个 Option 没有定义将返回另一个。当我们需要将可能的失败串联起来时，如果第一个失败尝试第二个，这会很有用。

一个常见的用法是 o.getOrElse(throw new Exception("FAIL"))。将结果是 None 的 Option 转回一个异常。一般的经验法则是在没有合理的方案能捕获异常时将其抛出；如果异常是一种可恢复的错误，使用 Option（或 Either，稍后讨论）会更加灵活。

如你所见，将错误作为一个普通的值返回便于我们使用高阶函数实现与使用异常相似的某种错误逻辑的合并。注意，不必在运算的每个阶段都检测 None，可以进行一些转换，等准备好了再检测和处理 None。除此之外还获得了一个好处：类型安全，因为 Option[A] 与 A 是不同的类型，编译器不会让我们忘记显式地推迟处理 None 的可能。

4.3.2 Option 的组合、提升及对面向异常的 API 的包装

一旦开始使用 Option，可能很容易产生一个看法：它对当前整个代码库都会造成影响。想象一下有多少方法的调用者接收或返回 Option，不得不修改成处理 Some 和 None 的情况。但这并不会发生，因为我们可以把普通函数提升（lift）为一个对 Option 操作的函数。

举一个例子，map 函数让我们用一个类型为 A=>B 的函数对类型为 Option[A] 的值进行操作，并返回 Option[B]。从另一个视角可以看作是 map 把一个 A=>B 的函数 f 变成类型为 Option[A]=>Option[B] 的函数。让这个看上去更明显一些：

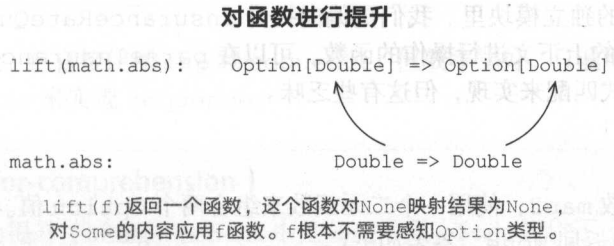
```
def lift[A,B](f: A => B): Option[A] => Option[B] = _ map f
```

也就是说任何已存在的普通函数都可以转换成（通过提升）在一个 Option 值的上下文里进行操作的函数。看一个例子：

```
val abs0: Option[Double] => Option[Double] = lift(math.abs)
```

math 对象包含 abs、sqrt、exp 等各种独立的数学函数。我们不需要重写 math.abs 函数来处理存在可选性（optional）的值，只需要将其提升到 Option 上下文。可以对任何函数都这么干。看另一个例子，假设我们要对一个汽车保险公司的网站实现一段逻辑，用户可以在一个页面提交表单请求一个即时在线报价。我们将从表单里解析信息，最后调用换算函数：

```
/**
 * 根据两个关键因子计算每年汽车保险费的绝密配方
 */
def insuranceRateQuote(age: Int, numberOfSpeedingTickets: Int): Double
```



我们希望能够调用刚定义的这个函数，不过用户从页面表单上所提交的年龄和超速罚款单号码都是字符串，需要将它们解析为整数，而解析可能会失败；对于给定的字符串 s 可以用 s.toInt 解析为整数，如果它不是一个有效的整数将抛出 NumberFormatException：

```
scala> "112".toInt
res0: Int = 112
```

```
scala> "hello".toInt
java.lang.NumberFormatException: For input string: "hello"
  at java.lang.NumberFormatException.forInputString(...)
  ...
```

让我们把 toInt 这个基于异常的 API 转换成 Option，看看是否可以实现一个 parseInsuranceRateQuote 函数，它以字符串方式接收年龄和超速罚款号码，如果解析成功再试图调用 insuranceRateQuote 函数。

示例 4.3 使用 Option

```
def parseInsuranceRateQuote(
        age: String,
        numberOfSpeedingTickets: String): Option[Double] = {
    val optAge: Option[Int] = Try(age.toInt)
    val optTickets: Option[Int] = Try(numberOfSpeedingTickets.toInt)
    insuranceRateQuote(optAge, optTickets)
}

def Try[A](a: => A): Option[A] =
    try Some(a)
    catch { case e: Exception => None }
```

toInt方法对任何字符串都可用。

没有类型检测！见下面的讨论。

接收一个非严格求值的A参数，我们可以在对a求值时捕获任何异常并转换为None。

注意：这会丢弃掉错误信息。我们会在本章后边改进。

Try 函数是一个通用目的的函数，用于将一个基于异常的 API 转换为一个面向 Option 的 API。参数 a 的类型 =>A 表示它是一个非严格（或惰性）求值参数。我们在下一章讨论更多有关惰性求值的内容。

但是这里有个问题——在我们解析 optAge 和 optTickets 为 Option[Int] 之后，如何调用接收两个 Int 参数的 insuranceRateQuote 函数？需要把 insuranceRateQuote 重写为接收 Option[Int] 吗？不，修改 insuranceRateQuote 会让关注点混乱，会强迫它意识到之前的运算可能会失败，更不必说我们可能没法去修改——它可能定义在一个我们无权修改的独立模块里。我们更偏向提升 insuranceRateQuote 为对两个存在可选值（optional）的上下文进行操作的函数。可以在 parseInsuranceRateQuote 方法体里使用显式的模式匹配来实现，但这有些乏味。

◈ 练习 4.3

写一个泛型函数 map2，使用一个二元函数来组合两个 Option 值。如果两个 Option 都为 None，也返回 None。签名如下：

```
def map2[A,B,C](a: Option[A], b: Option[B])(f: (A, B) => C): Option[C]
```

现在可以用 map2 来实现 parseInsuranceRateQuote：

```
def parseInsuranceRateQuote(
        age: String,
        numberOfSpeedingTickets: String): Option[Double] = {
    val optAge: Option[Int] = Try { age.toInt }
```

接收单个参数的函数，在调用时可以用大括号替换小括号；这里等价于 Try(age.toInt)。

```
    val optTickets: Option[Int] = Try { numberOfSpeedingTickets.toInt }
    map2(optAge, optTickets)(insuranceRateQuote)
}
```

如果任何一个解析失败，将立即返回None。

有了 map2 函数，意味着我们不需要修改任何已存在函数，让它们感知 Option。可以提升它们操作 Option 上下文。你是否已经学会定义 map3、map4、map5？再看看其他相似的例子。

◈ 练习 4.4

写一个 sequence 函数，将一个 Option 列表结合为一个 Option。这个结果 Option 包含原 Option 列表中的所有元素（用 Some 封装的）值。如果原 Option 列表中出现一个 None，函数结果也应该返回 None；否则结果应该是所有（使用 Some 包装的）元素值的列表。签名如下：[3]

```
def sequence[A](a: List[Option[A]]): Option[List[A]]
```

有时想对一个列表使用函数进行 map 操作，但这个函数执行可能会失败，如果列表中任何元素在应用这个函数时为 None 就返回 None。比如，有一个字符串列表，希望对每个字符串解析成 Option[Int]，那么整个列表解析的结果应该是什么？这种情况我们可以简单地对 map 操作的结果调用 sequence 函数：

```
def parseInts(a: List[String]): Option[List[Int]] =
    sequence(a map (i => Try(i.toInt)))
```

不幸的是这种方式效率有些差，因为它遍历了两次列表，第一次对每个字符串转换为 Option[Int]，第二次再对这些 Option[Int] 组合成 Option[List[Int]]。对 map 结果再进行 sequence 操作也是一种常见操作，也有必要实现一个新的泛型函数 traverse，签名如下：

```
def traverse[A, B](a: List[A])(f: A => Option[B]): Option[List[B]]
```

◈ 练习 4.5

实现一个函数，它直接使用 map 和 sequence，但效率更好，只遍历一次列表。实际上，按照 traverse 来实现 sequence。

For 推导（for-comprehension）

既然对函数的提升（lifting）在 Scala 中是如此常见，Scala 提供了一个叫作“for 推导”的语法结构，它会自动展开为一系列的 flatMap 和 map 调用。让我们看一下 map2 是如何用 for 推导来实现的。

先看原始实现代码：

3 这是一个不适合用面向对象风格定义函数的清晰的例子。这不应该是 List 的一个方法（List 不应该对 Option 有任何了解）也不能是 Option 的方法，所以它被放在了 Option 的伴生对象里。

```
def map2[A,B,C](a: Option[A], b: Option[B])(f: (A, B) => C): Option[C] =
    a flatMap (aa =>
    b map (bb =>
    f(aa, bb)))
```

下面是使用 for 推导实现的版本：

```
def map2[A,B,C](a: Option[A], b: Option[B])(f: (A, B) => C): Option[C] =
    for {
        aa <- a
        bb <- b
    } yield f(aa, bb)
```

for 推导由一系列例如 `aa <- a` 这样的绑定（binding）组成，在右大括号后边跟着 `yield` 关键字，`yield` 可以使用先前在 `<-` 符号左边绑定的任何值。编译器会对这些绑定操作的语法糖转换为 `flatMap` 调用，对最后一个绑定和 `yield` 会转换为 `map` 调用。

你可以在任何显式调用 `flatMap` 和 `map` 的地方用 for 推导来代替。

有了 `map`、`lift`、`sequence`、`traverse`、`map2`、`map3` 等方法，对任何已存在的函数不必修改就可以使它们对可选值（optional value）也能执行操作。

4.4 Either 数据类型

这一章的核心概念是我们可以用普通的值类表现失败和异常，将对错误处理和恢复的通用模式抽象出来用函数实现。`Option` 不是用于这种目的的唯一数据类型，虽然使用得很频繁，但它有些过于简单。有一点或许你已经注意到，`Option` 不会告诉我们在异常条件下发生了什么错误，它只是给我们一个 `None`，表示没有可用的值。但有时我们想要知道更多，比如想要一个字符串给出更多信息或者当异常发生时想知道实际错误是什么。

可以创建一个数据类型，能对我们想要的失败信息进行编码。有些时候只需要知道是否发生了失败，这种情况可以使用 `Option`，而有时则想知道更多信息。这一节我们将示范一个对 `Option` 的简单扩展：`Either` 数据类型，它可以跟踪失败原因。看一下它的定义：

```
sealed trait Either[+E, +A]
case class Left[+E](value: E) extends Either[E, Nothing]
case class Right[+A](value: A) extends Either[Nothing, A]
```

就像 `Option`，`Either` 只有两种情况。它们的实质区别是 `Either` 的两种情况都有值。Either 数据类型是一种常见类型，它的值是两种情况中的一种。我们称它是两个类型的“*互斥并集*”（disjoint union）。当我们用它来表示成功或失败时，习惯上 `Right` 构造器用于表示成功（right 这里有双关的意思，既表示右边也表示正确），`Left` 构造器用于表示失败。建议给 `Left` 的类型参数命名为 `E`（代表*错误*）。[4]

4 Either 也是经常被广泛使用在编写一个具有两种可能性但又不值得为它定义一个新的数据类型的场景。我们会在整本书中看到一些这样的例子。

标准库中的 Option 和 Either

在本章前边提到过，Option 和 Either 也存在于 Scala 的标准库中（Option API 文档 http://mng.bz/fiJ5；Either API 文档 http:// mng.bz/106L），我们这里定义的大部分函数在标准库的版本中也存在。

建议你阅读一下标准库中 Option 和 Either API 的相关内容，对比它们的差异，尽管有几个缺失的函数，尤其是 sequence、traverse 和 map3，Either 也没有我们这里定义的“偏向右侧操作”的 flatMap 函数。标准库中的 Either 稍微复杂一些，更多细节请阅读 API 文档。

再看一下 mean 函数的例子，这次在失败的情况下返回一个字符串：

```
def mean(xs: IndexedSeq[Double]): Either[String, Double] =
    if (xs.isEmpty)
        Left("mean of empty list!")
    else
        Right(xs.sum / xs.length)
```

有时我们想要对错误包含更多的信息，比如堆栈调用信息，以便在源码中定位错误。这种情况下也可以对 Either 的 Left 返回一个异常：

```
def safeDiv(x: Int, y: Int): Either[Exception, Int] =
    try Right(x / y)
    catch { case e: Exception => Left(e) }
```

我们之前用 Option 实现的用于将异常转换为值这一通用模式的 Try 函数，也可以用 Either 来实现：

```
def Try[A](a: => A): Either[Exception, A] =
    try Right(a)
    catch { case e: Exception => Left(e) }
```

◈ **练习 4.6**

实现 Either 版本的 map、flatMap、orElse 和 map2 函数。

```
trait Either[+E, +A] {
    def map[B](f: A => B): Either[E, B]
    def flatMap[EE >: E, B](f: A => Either[EE, B]): Either[EE, B]
    def orElse[EE >: E,B >: A](b: => Either[EE, B]): Either[EE, B]
    def map2[EE >: E, B, C](b: Either[EE, B])(f: (A, B) => C): Either[EE, C]
}
```

对右侧进行mapping时，必须将左边的类型参数提升为父类型（supertype）以满足+E型变注释。

orElse跟上面类似。

注意，有了这些定义，Either 现在也可以使用 for 推导，例如：

```
def parseInsuranceRateQuote(
    age: String,
    numberOfSpeedingTickets: String): Either[Exception,Double] =
```

```
for {
    a <- Try { age.toInt }
    tickets <- Try { numberOfSpeedingTickes.toInt }
} yield insuranceRateQuote(a, tickets)
```

现在我们对发生的异常给出了更多信息而不是只在失败时得到 None。

◈ 练习 4.7

对 Either 实现 sequence 和 traverse，如果遇到错误返回第一个错误。

```
def sequence[E, A](es: List[Either[E, A]]): Either[E, List[A]]

def traverse[E, A, B](as: List[A])(
                      f: A => Either[E, B]): Either[E, List[B]]
```

最后一个例子，这儿有一个 map2 的应用，用于 mkPerson 函数在构造一个有效的 Person 之前校验 name 和 age。

示例 4.4 使用 Either 校验数据

```
case class Person(name: Name, age: Age)
sealed class Name(val value: String)
sealed class Age(val value: Int)

def mkName(name: String): Either[String, Name] =
    if (name == "" || name == null) Left("Name is empty.")
    else Right(new Name(name))

def mkAge(age: Int): Either[String, Age] =
    if (age < 0) Left("Age is out of range.")
    else Right(new Age(age))

def mkPerson(name: String, age: Int): Either[String, Person] =
    mkName(name).map2(mkAge(age))(Person(_, _))
```

◈ 练习 4.8

在这个实现里，即使 name 和 age 都无效，map2 也只能报出一个错误。为了让两个错误都能报出来，你需要做些什么改变？你会改变 map2 或 mkPerson 的签名吗？或者你会通过一些辅助结构创建一种新的数据类型比 Either 更好地满足这一需求吗？这种更好的数据类型的 orElse、traverse 和 sequence 行为与 Either 有何不同？

4.5 小结

在这一章我们注意到了使用异常存在的一些问题，并引入了纯函数式错误处理的基本原则。虽然焦点集中在 Option 和 Either 等代数数据类型，核心思想还是用普通值来表

现异常，将错误处理和传播的通用模式使用高阶函数封装起来。把作用（effect）当作值来表述，我们会在本书不断地看到这样的各种例子。

不期望你能流畅地使用本章写的所有高阶函数，但应该足够熟悉，可以开始完整地写带有错误处理的函数代码。有了这些新工具，异常应该保留在只有不可恢复的情况下才使用。

最后，在这一章我们简单接触了一下非严格求值函数的概念（回忆一下 `orElse`、`getOrElse` 和 `Try` 函数），在下一章我们将更近距离地观察为什么非严格函数这么重要，它如何在函数式编程中带来更大的模块化和更高的效率。

5 严格求值和惰性求值

在第 3 章我们谈论过函数式数据结构，并用一个单向链表作为例子。覆盖了一堆在列表上的操作——map、filter、foldLeft、foldRight 和 zipWith 等。我们注意到这些操作都有输入参数，并输出一个新构造的列表。

想象一下你有一副扑克，要去除奇数扑克，然后翻出所有的 Q。理想情况是只需要传入单个参数给这副扑克，这个参数能同时查找 Q 和奇数。这比删除奇数扑克然后再在剩余扑克里找 Q 更高效。后一种方式在 Scala 中用下面的代码：[1]

```
scala> List(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).map(_ * 3)
List(36,42)
```

上面的表达式中 map(_+10) 会产生一个中间列表，然后再对这个中间列表进行 filter(_%2==0) 操作，这个 filter 操作也会产生一个列表，接下来再进行 map(_*3) 操作产生最终的列表。换句话说每一次转换都产生了一个临时列表作为下一次转换的输入，之后便被立即丢弃。

考虑一下这段程序是如何求值的，如果我们手动跟踪一下求值过程，步骤如下：

示例 5.1 跟踪 List 程序

```
List(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).map(_ * 3)

List(11,12,13,14).filter(_ % 2 == 0).map(_ * 3)

List(12,14).map(_ * 3)

List(36,42)
```

这里我们展示了对表达式每次执行求值结果的替代。比如从第一行到第二行，根据 map 的定义把 List(1,2,3,4).map(_+10) 替换为了 List(11,12,13,14)。[2] 这个视

1 我们现在使用 Scala 标注库里的 List 类型的 map 和 filter 都是 List 里的方法，而不像第 3 章里我们写的独立的函数。

2 像这样跟踪程序，往往更直观，不用完整地跟踪每一个子表达式的求值。在这个例子里，我们忽略了 List(1,2,3,4).map(_ + 10) 表达式的完整的展开方式。我们可以进入 map 的定义并跟踪它的执行过程，但这里我们忽略这些细节。

图清晰地展示了怎么调用 map 和 filter 的，每次遍历自身，对每个元素使用输入的函数进行求值，并输出一个新分配的列表。那么如果我们对一系列的转换融合成单个函数而避免产生临时数据结构岂不更好？可以在一个 while 循环里手工重写这段代码，但理想情况是能够自动实现，并保留高阶组合风格。我们想使用高阶函数如 map、filter 组合我们的程序而不是在单个大循环体里实现。

我们可以通过使用非严格求值 (non-strictness) 来实现这种自动循环的融合物。在这一章将解释非严格求值（非正式的说法：惰性求值）是怎么回事，并实现一个融合了一系列转换的惰性 List 类型。虽然本章的动机只是构造一个更好的 List，但我们在这一过程中将看到非严格求值是一项提升函数式编程效率和模块化的基础技术。

5.1 严格和非严格函数

在我们实现惰性 List 之前，需要了解一些基础。什么是严格求值和非严格求值？这些概念在 Scala 中是怎么表达的？

非严格求值是函数的一种属性，称一个函数是非严格求值的意思是这个函数可以选择不对它的一个或多个参数求值。相反，一个严格求值的函数总是对它的参数求值。严格求值函数在大部分编程语言中是一种常态。并且大部分编程语言也只支持严格求值。在 Scala 中除非明确声明，否则任何函数都是严格求值的（到目前为止我们所定义过的函数都是严格的）。看一个例子：

```
def square(x: Double): Double = x * x
```

当你调用 square(41.0 + 1.0) 时，square 函数将接收到完成求值的 42.0 作为参数，因为它是严格求值的。如果你调用 square(sys.error("failure"))，将在 square 方法执行之前先得到一个异常，因为 sys.error("failure") 表达式会先被求值。[3]

虽然我们还没有学习任何表示 Scala 非严格求值的语法，但你无疑会对这个概念有些熟悉。举例来说，在很多编程语言包含 Scala 中都出现的短路 Boolean 函数 && 和 ||，就是非严格求值。你或许习惯于把 && 和 || 当成语言的内置语法，其实也可以当成可以选择不对参数求值的函数。&& 函数接收 2 个 Boolean 参数，但只有第一个参数是 true 的情况下才对第二个参数求值。

```
scala> false && { println("!!"); true } // 不打印任何内容
res0: Boolean = false
```

|| 函数只有在第一个参数为 false 的情况下才对第二个参数求值。

```
scala> true || { println("!!"); false } // 同样不打印任何内容
res1: Boolean = true
```

另一个非严格求值的例子是 Scala 中的 if 控制结构：

```
val result = if (input.isEmpty) sys.error("empty input") else input
```

3 sys.error 方法会抛出异常。——译者注

尽管 if 是 Scala 语言的内置结构，也可以看作是一个接收 3 个参数的函数：一个 Boolean 类型的条件参数；一个返回类型为 A 的表达式在条件为 true 时执行；另一个返回类型也为 A 的表达式在条件为 false 时执行。这个 if 函数也是非严格求值的。因为它不会对所有的参数都求值。更精确地说，if 函数对条件参数是严格求值的，因为它总要对条件判断来决定选择哪个分支；对两个 true 和 false 的分支参数是非严格求值的，因为它只对其中满足条件的一个参数求值。

在 Scala 中我们可以通过接收某个未求值的参数来写非严格求值函数。先展示这种方式，然后再展示更好的 Scala 内置的语法。下面是一个非严格求值的 if 函数：

```
def if2[A](cond: Boolean, onTrue: () => A, onFalse: () => A): A =
    if (cond) onTrue() else onFalse()

if2(a < 22,
    () => println("a"), ←——函数字面量语法，创建一个() => A的函数。
    () => println("b")
)
```

我们传入的未求值的参数类型前边紧接着一个 ()=> 符号。()=>A 类型的值表示一个函数，接收 0 个参数并返回一个 A 类型。[4] 通常一个表达式的未求值形式称为 *thunk*，我们可以强制对 thunk 求值得到结果。通过传入一个空参数列表调用函数，如 onTrue() 或 onFalse()。同样 if2 的调用者必须显式地创建 thunk，语法与之前我们见过的函数字面量（literal）相同。

总的来说，这种语法也很清晰——我们在每一个非严格求值参数的地方传入一个无参函数，然后在方法体里显式地调用这个函数获取结果。但这是一个非常常见的例子，所以 Scala 提供了一些更好的语法：

```
def if2[A](cond: Boolean, onTrue: => A, onFalse: => A): A =
    if (cond) onTrue else onFalse
```

我们传递的未求值的参数有一个箭头 => 紧接在类型前边。在方法体中我们不需要对使用 => 标注的参数做任何事情，就像往常一样只引用标识符也不需要对这个函数做任何特殊调用，只需要使用正常的函数调用语法。Scala 会负责为我们将表达式包装为 thunk。

```
scala> if2(false, sys.error("fail"), 3)
res2: Int = 3
```

使用两者中的任何一种语法，给一个函数传递一个未求值的参数，在方法体中引用的地方会被求值一次。即 Scala 不会（默认）缓存一个参数的求值结果：

```
scala> def maybeTwice(b: Boolean, i: => Int) = if (b) i+i else 0
maybeTwice: (b: Boolean, i: => Int)Int

scala> val x = maybeTwice(true, { println("hi"); 1+41 })
hi
hi
x: Int = 84
```

4 实际上 ()=>A 类型是 Function0[A] 类型的语法别名。

这里 i 在 maybeTwice 方法体中引用了两次，我们传入了一个代码块 {println("hi"); 1+41}，让它的求值过程更明显一些，它会以副作用方式打印 "hi"，在返回 42 这个结果之前表达式 1+41 也将被计算两次。如果我们希望只求值一次，可以通过 lazy 关键字显式地缓存这个值：

```
scala> def maybeTwice2(b: Boolean, i: => Int) = {
     |   lazy val j = i
     |   if (b) j+j else 0
     | }
maybeTwice: (b: Boolean, i: => Int)Int

scala> val x = maybeTwice2(true, { println("hi"); 1+41 })
hi
x: Int = 84
```

对一个 val 声明的变量添加 lazy 修饰符，将导致 Scala 延迟对这个变量求值，直到它第一次被引用的时候。它也会缓存结果，在后续引用的地方不会触发重复求值。

严格求值的正式定义

如果对一个表达式的求值一直运行或抛出一个错误而非返回一个定义的值，我们说这个表达式没有结束（terminate），或者说它是 evaluates to bottom。[5] 如果表达式 f(x) 对所有的 evaluates to bottom 的表达是 x，也是 evaluates to bottom，那么 f 是严格求值的。

最后一点术语，Scala 中非严格求值的函数接收的参数是传名参数（by name）而非传值参数（by value）。

5.2 一个扩展例子：惰性列表

现在回顾一下本章最初提出的问题。我们将使用*惰性列表*（lazy list）或 *Stream* 作为一个例子探索惰性化在函数式编程中是如何用于提升效率和模块化的。我们将看到基于流（Stream）的转换链怎么通过使用惰性化融合成一次操作。这里是一个简单的 Stream 的定义，接下来会讨论一些新的东西。

示例 5.2 Stream 的简单定义

```
sealed trait Stream[+A]
case object Empty extends Stream[Nothing]
case class Cons[+A](h: () => A, t: () => Stream[A]) extends Stream[A]
```

一个非空的stream由head和tail组成，head和tail都是非严格求值的。因为技术限制，这些参数都是必须明确强制求值的thunk，而非传名参数。

5 evaluate to bottom 是指那些非正常返回值的表达式，比如抛出异常、卡在循环里或停止程序等，bottom 这个词源自于形式逻辑。——译者注

```
object Stream {
    def cons[A](hd: => A, tl: => Stream[A]): Stream[A] = {
        lazy val head = hd
        lazy val tail = tl
        Cons(() => head, () => tail)
    }
    def empty[A]: Stream[A] = Empty
    def apply[A](as: A*): Stream[A] =
        if (as.isEmpty) empty else cons(as.head, apply(as.tail: _*))
}
```

用于创建空stream的智能构造器。

对惰性求值的head和tail做缓存，避免重复求值。

用于创建特定类型stream的智能构造器。

一个便利的可变参方法，用于根据多个元素构建一个Stream。

这个类型看上去跟 List 类型完全相同，除了 Cons 数据构造器接收显式的 thunk（() => A 和 () => Stream[A]）而非常规值。如果我们希望检测或遍历 Stream，需要强制这些 thunk 按照我们先前定义的 if2 的方式。举例来说，下面是一个以可选的方式抽取 Stream 的 head 的函数：

```
def headOption: Option[A] = this match {
    case Empty => None
    case Cons(h, t) => Some(h())
}
```

对h thunk显式地调用h()强制求值。

注意，我们必须显式调用 h() 来对 h 强制求值，不这样做代码会跟 List 一样。但 Stream 直到这部分真正需要时才求值（我们不会对 Cons 的 tail 求值）的能力很有用。

5.2.1 对 Stream 保持记忆，避免重复运算

我们希望能缓存 Cons 节点的值，一旦它们被强制求值。如果我们直接用 Cons 数据构造器，比如下面的代码，实际运算了 2 次 expensive(x)：

```
val x = Cons(() => expensive(x), tl)
val h1 = x.headOption
val h2 = x.headOption
```

我们通常定义更智能（smart）的构造器来避免这个问题，它是一种构造数据类型的函数，同时能够保证一些附加的不变式或提供与真正构造签名稍微不同的用于模式匹配的能力。智能 (smart) 构造器的写法习惯上跟普通的数据构造器相似，但首字母为小写。[6] 这里智能的 cons 构造器负责将传名参数记录到 head 和 tail，这是一个常见的技巧。它确保 thunk 只运行一次，只在第一次使用时被强制求值。后续的调用会返回已缓存的 lazy val：

```
def cons[A](hd: => A, tl: => Stream[A]): Stream[A] = {
    lazy val head = hd
    lazy val tail = tl
    Cons(() => head, () => tail)
}
```

6 智能构造器并不是构造器而是普通方法。——译者注

空的智能构造器仅仅返回空（Empty），但 Empty 被标识为 Stream[A] 在某些情况下是一个更好的类型接口。[7] 我们看一下智能构造器在 Stream.apply 函数中是如何被使用的：

```
def apply[A](as: A*): Stream[A] =
    if (as.isEmpty) empty
    else cons(as.head, apply(as.tail: _*))
```

此外，Scala 负责包装 thunk 中传给 cons 的参数，所以 as.head 和 apply(as.tail: _*) 表达式不会求值，直到在 Stream 中被强制求值。

5.2.2 用于检测 Stream 的 helper 函数

在继续之前，让我们写一些 helper 函数使检测 Stream 更容易。

◈ **练习 5.1**

写一个可将 Stream 转换成 List 的函数，它会被强制求值，可以在 REPL 下看到值的内容。可以转换成标准库中的常规 List 类型，可以把这个函数以及其他操作 Stream 的函数放到 Stream 特质内部。

```
def toList: List[A]
```

◈ **练习 5.2**

写一个函数 take(n) 返回 Stream 中的前 *n* 个元素；写一个函数 drop(n) 返回 Stream 中第 *n* 个元素之后的所有元素。

◈ **练习 5.3**

写一个函数 takeWhile 返回 Stream 中从起始元素连续满足给定断言的所有元素。

```
def takeWhile(p: A => Boolean): Stream[A]
```

你可以在 REPL 下使用 take 和 toList 一起来检测 Stream，例如试着打印 Stream(1,2,3).take(2).toList。

5.3 把函数的描述与求值分离

函数式编程的主题之一是*关注分离*（separation of concerns）。我们希望将计算的描述与实际运行分开。在前几章我们已经以不同的方式接触过这一主题了，比如一等函数，[8] 捕获函数体内的运算逻辑，只有在接收到参数时才执行它。我们使用 Option 捕获实际发生

7 回想一下 Scala 使用子类型来表示数据构造器，但我们几乎总想要推断为 Stream 类型，而不是 Cons 或 Empty 类型。让智能构造器返回基类类型是一种常见的伎俩。

8 即函数对象与值一样都是一等公民。——译者注

的错误，而决定对它做什么是一个分离的关注点。对 Stream，可以构建一个产生一系列元素的计算逻辑直到实际需要这些元素时才运行。

一般而言，惰性化让我们对一个表达式分离了它的描述和求值。这给我们带来一种强大的能力——可以选择描述一个比我们所需要的更大的表达式，只对这个表达式的一部分求值。用一个例子来看，函数 exists 检查 Stream 中是否存在元素匹配一个 Boolean 函数：

```
def exists(p: A => Boolean): Boolean = this match {
    case Cons(h, t) => p(h()) || t().exists(p)
    case _ => false
}
```

注意 || 对它的第二个参数是非严格求值。如果 p(h()) 返回 true 那么 exists 提前终止遍历也返回 true。另外，Stream 的 tail 部分也是一个 lazy val，不仅仅是提前终止，Stream 的 tail 压根就没有被求值。所以不管 tail 里是什么代码，它根本没执行。

exists 函数使用显式的递归实现。回顾一下第 3 章中的 List 我们可以以 foldRight 方式实现一个更通用的递归，可以以惰性的方式对 Stream 做相同的事：

```
def foldRight[B](z: => B)(f: (A, => B) => B): B =
    this match {
        case Cons(h,t) => f(h(), t().foldRight(z)(f))
        case _ => z
    }
```

f参数的类型声明中B类型前边箭头=>表示第二个参数是传名参数，f不会对它进行求值。

如果f不对第二个参数求值，递归就不会发生。

这看上去与 List 中的 foldRight 非常相似，但注意我们的组合函数 f 对它的第二个参数是非严格求值的。如果 f 选择不对第二个参数求值，这会提前终止遍历。我们看一下使用 foldRight 实现的 exists：[9]

```
def exists(p: A => Boolean): Boolean =
    foldRight(false)((a, b) => p(a) || b)
```

这里 b 是对 Stream 的 tail 进行 fold 的未求值的递归步骤，如果 p(a) 返回 true，b 不会被求值，计算提前终止。

既然 foldRight 能提前终止遍历，我们可以重用它实现 exists。不能用严格求值版本的 foldRight 来实现，得写一个特定的 exists 递归函数来处理提前终止。惰性化让代码更容易复用。

◈ 练习 5.4

实现一个 forAll 函数，检查 Stream 中所有元素是否与给定的断言匹配。遇到不匹配的值应该立即终止遍历。

```
def forAll(p: A => Boolean): Boolean
```

◈ 练习 5.5

使用 foldRight 实现 takeWhile。

9 exists 的定义虽然直观，但是如果 stream 很大并且所有元素测试为 false（可能导致栈溢出），却不是栈安全的。

◈ 练习 5.6

难：使用 foldRight 实现 headOption。

◈ 练习 5.7

用 foldRight 实现 map、filter、append 和 flatMap，append 方法参数应该是非严格求值的。

注意这些实现是增量的（incremental）——它们不会生成完全的答案，直到某些其他计算看到由实际计算中生成的作为结果的 Stream 中的元素之后，才可以生成想要的元素。因为这种增量性质，我们可以一个接一个地调用这些函数而不用对中间结果实例化（fully instantiating）。

让我们看一个简单的程序，跟踪一段本章开头用于启发的例子：Stream(1,2,3,4).map(_ + 10).filter(_ % 2 == 0)。我们将转换这个表达式为一个 List 来强制求值。花几分钟跟踪并理解发生了什么。这比我们之前跟踪过的例子更有挑战一些。像这种惰性运算的跟踪过程，不过是相同的表达式一再地重复，每次求值都在上次的基础上更进一步。

示例 5.3 跟踪 Stream 程序

```
Stream(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).toList

cons(11, Stream(2,3,4).map(_ + 10)).filter(_ % 2 == 0).toList ←┐
                                                  对第1个元素应用map。
Stream(2,3,4).map(_ + 10).filter(_ % 2 == 0).toList ←对第1个元素应用filter。

cons(12, Stream(3,4).map(_ + 10)).filter(_ % 2 == 0).toList ←对第2个元素应用map。

12 :: Stream(3,4).map(_ + 10).filter(_ % 2 == 0).toList ←┐
                                  对第2个元素应用filter生成第1个元素结果。
12 :: cons(13, Stream(4).map(_ + 10)).filter(_ % 2 == 0).toList

12 :: Stream(4).map(_ + 10).filter(_ % 2 == 0).toList

12 :: cons(14, Stream().map(_ + 10)).filter(_ % 2 == 0).toList

12 :: 14 :: Stream().map(_ + 10).filter(_ % 2 == 0).toList ←┐
                                  对第4个元素应用filter并生成结果元素。
12 :: 14 :: List() ←map和filter没有更多要做的了，空stream变成了空list。
```

跟踪过程中要注意的是 filter 和 map 转换是如何交错的——计算在用于生成单个元素的 map 函数与用于测试元素是否被 2 整除的 filter 函数之间交替进行。注意我们不会完全实例化 map 运算结果的中间 Stream，就跟我们使用一个专用的循环处理交错逻辑一样。因为这个原因，人们有时也把 Stream 描述为“一等循环”（first-class loops），这些逻辑可以使用高阶函数，如 map、filter 等组合。

既然中间 Stream 不会实例化，很容易以全新的方式复用已存在的组合子，而不用担心我们对 Stream 处理超出需要的部分。举个例子，可以复用 `filter` 来定义 `find`，一个方法返回第一个匹配的元素，如果存在的话。虽然 `filter` 转换了整个 stream，但转换是惰性的，所以 `find` 一遇到匹配的元素就立刻终止：

```
def find(p: A => Boolean): Option[A] =
  filter(p).headOption
```

Stream 转换的增量性质对内存使用也有重要的影响。因为不再生成中间 Stream，转换只需要处理当前元素所够用的内存。比如，`Stream(1,2,3,4).map(_ + 10).filter(_ % 2 == 0)` 这个转换中垃圾收集会回收分配给由 `map` 产生的值 11 和 13 的内存空间，当 `filter` 决定它们不再被需要时。当然这是一个简单的例子，某些场景我们可能要处理大量元素，而且元素自身也可能是一个占用内存明显的大对象，能够尽快回收这些内存可以降低程序对内存的需求。

关于定义提升内存效率的流式计算（streaming calculation）还有很多要说的，特别是需要 I/O 的计算，会在本书第四部分介绍。

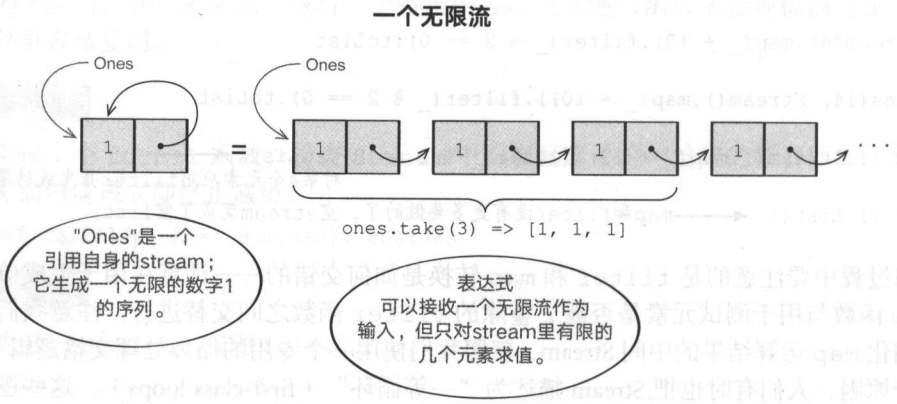
5.4 无限流与共递归

正因为具有增量性质，我们所写的函数也适用于*无限流*（infinite stream），这里是一个由 1 组成的无限流的例子：

```
val ones: Stream[Int] = Stream.cons(1, ones)
```

尽管 `ones` 是无限的，我们目前为止所写的函数只检测了 Stream 必要的部分，生成需要的输出，看一个例子：

```
scala> ones.take(5).toList
res0: List[Int] = List(1, 1, 1, 1, 1)

scala> ones.exists(_ % 2 != 0)
res1: Boolean = true
```



很多函数可以对有限资源求值，甚至它们的输入产生的是一个无限序列。

再来几个例子：

- `ones.map(_ + 1).exists(_ % 2 == 0)`
- `ones.takeWhile(_ == 1)`
- `ones.forAll(_ != 1)`

在每个例子里，我们立即取得一个结果。不过要小心，因为很容易写出永不结束或线程栈不安全的表达式。比如 `ones.forAll(_ == 1)` 将一直检测后续的元素，因为不会遇到一个可以让它终止遍历并给出确切答案的元素（最终会导致堆栈溢出而非无限循环）。[10]

让我们看看还有什么其他可以生成 Stream 的函数。

◈ 练习 5.8

对 `ones` 稍微泛化一下，定义一个 `constant` 函数，根据给定值返回一个无限流。

```
def constant[A](a: A): Stream[A]
```

◈ 练习 5.9

写一个函数生成一个整数无限流，从 `n` 开始，然后 `n+1`、`n+2`，等等。[11]

```
def from(n: Int): Stream[Int]
```

◈ 练习 5.10

写一个 `fibs` 函数生成斐波那契数列的无限流：0, 1, 1, 2, 3, 5, 8，等等。

◈ 练习 5.11

写一个更加通用的构造流的函数 `unfold`。它接收一个初始状态，以及一个在生成的 Stream 中用于产生下一状态和下一个值的函数。

```
def unfold[A, S](z: S)(f: S => Option[(A, S)]): Stream[A]
```

`Option` 用于表示 `Stream` 何时结束，如果真的发生的话。`unfold` 是一个非常通用的流构造函数。

`unfold` 函数是一个被称作*共递归*（corecursive）函数的例子。递归函数由不断地对更小范围的输入参数进行递归调用而结束（terminate）；而共递归函数只要保持生产数据不需要结束，这意味着我们总是可以在一个有限的时间段里对更多的结果求值。`unfold` 函数生产能力随 `f` 函数的结束而结束，因为我们只需要再次运行 `f` 函数生成 Stream 中的下一个元素。共递归有时也被称为*守护递归*（guarded recursion），生产能力有时也被称为*共结*

10 使用普通的递归循环定义一个栈安全版本的 `forAll` 也是可能的。

11 在 Scala 里 Int 类型是一个有符号的 32 位整型，所以 stream 会在某个点上从正数变为负数，并且在大约 40 亿元素后重复发生。

束（cotermination）。这些术语对我们的讨论不是很重要，但有时会在函数式编程的上下文里看到它们，如果好奇它们源自哪里，想了解得更深一些的话可以从本章的注释中去获取。

◈ 练习 5.12

根据 unfold 函数来写 fibs、from、constant 和 ones。[12]

◈ 练习 5.13

使用 unfold 实现 map、take、takeWhile、zipWith（如第 3 章）以及 zipAll。zipAll 函数应该继续遍历只要 Stream 还有更多元素——它使用 Option 表示 Stream 是否已经彻底遍历完了。

```
def zipAll[B](s2: Stream[B]): Stream[(Option[A],Option[B])]
```

现在我们已经练习写了一些 Stream 函数，让我们回到第 3 章结尾的练习——hasSubsequence 函数，检测是否一个 list 包含一个给定子序列。在使用严格求值的 List 结构和 List 操作函数时，被迫写了一个非常复杂的大循环来实现，使用 lazy List 你能看出如何通过组合一些已经写好的函数就可以实现 hasSubsequence 吗？[13] 在继续之前，先试着考虑一下自己的方式。

◈ 练习 5.14

难：使用已写过的函数实现 startsWith 函数。它检查一个 Stream 是否是另一个 Stream 的前缀，比如 Stream(1,2,3) startsWith Stream(1,2) 返回 true。

```
def startsWith[A](s: Stream[A]): Boolean
```

◈ 练习 5.15

使用 unfold 实现 tails 函数，对一个给定的 Stream，tails 返回这个 Stream 输入序列的所有后缀（包含原始 Stream），比如给定 Stream(1,2,3) 返回 Stream(Stream(1,2,3), Stream(2,3), Stream(3), Stream())。

```
def tails: Stream[Stream[A]]
```

现在可以用之前写过的函数来实现 hasSubsequence：

```
def hasSubsequence[A](s: Stream[A]): Boolean =
  tails exists (_ startsWith s)
```

---

12 使用 unfold 来定义 constant 和 ones 意味着我们没有得到共享，作为递归定义 val ones: Stream[Int] = cons(1, ones)。递归定义消费常量的内存，即使我们在遍历时保持引用，而基于 unfold 的实现则不会。保留共享不是我们使用 stream 编程时通常依赖，因为它非常微妙，并且不是由类型跟踪的。例如，当调用 xs.map(x => x) 共享被破坏掉。

13 这个用惰性化方式将更小的函数组装成 hasSubsequence 的小例子来自 Cale Gibbard。参考：http://lambda-the-ultimate.org/node/1277#comment-14313。

这个实现与之前的一大段使用嵌套循环并允许逻辑可以提前跳出每一层循环的实现方式有相同的执行步骤。通过使用惰性化我们可以用更简单的组件来组合函数，并与特定实现（更烦琐的）保持相近的效率。

◇ **练习 5.16**

难：把 `tails` 泛化为 `scanRight` 函数，类似 `foldRight` 返回一个中间结果的 Stream，例如：

```
scala> Stream(1,2,3).scanRight(0)(_ + _).toList
res0: List[Int] = List(6,5,3,0)
```

这个例子应该等同于表达式 `List(1+2+3+0, 2+3+0, 3+0, 0)`。你的函数应该复用中间结果，这样遍历一个 `Stream` 的 `n` 个元素耗时总是线性 `n`。能用 `unfold` 实现吗？怎么实现或者为什么不能？能用我们写过的其他函数实现吗？

5.5 小结

在这一章，我们介绍了非严格求值。作为提升函数式编程效率和模块化的基本技能。非严格求值可以认为是用于弥补函数式代码效率的一种技术，但它还有一个更好的意图——非严格求值通过分离表达式的描述和求值同时提升了模块化。保持关注分离，让我们在多个上下文复用描述，对不同部分表达式的求值获得不同的结果。如果像严格求值那样将描述和求值纠缠在一起将无法做到。在本章课程里我们看到基于关注分离这一原则的一些例子，在本书剩余部分还会看到更多例子。

我们在下一章将切换到函数式状态的实现。这是我们开始探索函数式设计过程之前的最后一道障碍。

6 纯函数式状态

在这一章，我们将看到如何以纯函数式编程来操作状态，用一个最简单的随机数生成（random number generation）做例子，虽然它不是这一章最引人注目的用例，但随机数的简单性很适合做第一个例子。我们会在第三和第四部分看到更多引人注目的用例。特别是第四部分，会讨论很多对状态和作用（effect）的处理。这里的目标只是给你一些能够以纯函数式实现有状态的API的基本模式。你可能已经遇到过很多跟这里要探索的相似的问题。

6.1 以副作用方式生成随机数

如果你需要在Scala里生成随机[1]数，在标准库里就有一个类`scala.util.Random`，[2]它是一个典型的依赖副作用的命令式API。这是一个使用的例子：

示例 6.1 使用 scala.util.Random 生成随机数

```
scala> val rng = new scala.util.Random ←——使用当前时间创建一个随机数种子。

scala> rng.nextDouble
res1: Double = 0.9867076608154569

scala> rng.nextDouble
res2: Double = 0.8455696498024141

scala> rng.nextInt
res3: Int = -623297295

scala> rng.nextInt(10) ←——返回0到9之间的一个随机数。
res4: Int = 4
```

即使我们对`scala.util.Random`会发生什么一无所知，我们可以假定`rng`对象有一些内部状态能在每次调用之后更新，否则每次调用`nextInt`或`nextDouble`应该得到相同的值。因为状态的更新以副作用的方式执行，这些方法不是引用透明的。这意味着它是难以测试、组合、模块化和并行化的。

---

1 实际上是伪随机数，但我们先忽略这个区别。

2 Scala API链接：http://mng.bz/3DP7。

我们选择可测试性作为例子。如果我们想写一个使用随机数的方法，需要测试它是可再现的。比如说我们有一个副作用的方法，目的是模拟投 6 面色子死亡法，也就是返回 1 到 6 之间的一个数（包含 1 和 6）：

```
def rollDie: Int = {
  val rng = new scala.util.Random
  rng.nextInt(6)  ←——返回0~5之间的随机数。
}
```

这个方法有一个偏差为 1 的错误，假定返回值是 1 到 6，实现返回值是 0 到 5。尽管如此它还是无法正常工作，6 次测试中有 5 次将遇到的是特定值！如果测试失败，若我们能可靠地重现失败，它将是不切实际的。

注意这里重要的不是这个特殊的例子，而是概念。这个例子里 bug 很明显，也很容易重现。但很容易想象模拟一个更加复杂的方法，bug 也更微妙。程序越复杂 bug 也更难以察觉，能够稳定地重现 bug 就更加重要。

一个建议是将随机数生成器传给方法。这样的话当我们想在测试里重现失败时，我们可以传入导致失败的相同的生成器：

```
def rollDie(rng: scala.util.Random): Int = rng.nextInt(6)
```

但是这个解决方式也有一个问题。“相同”的生成器必须使用相同的随机数种子（seed）创建，也是相同的状态。这意味着这个“相同”的生成器的方法也要被调用确定的次数，这很难保证，因为每次调用 `nextInt` 随机数生成器的前一个状态已经销毁！难道我们需要一个独立的机制跟踪我们对 `Random` 调用了多少次？

答案是否定的，我们应该根据原则避开副作用。

6.2 纯函数式随机数生成器

恢复引用透明的关键是让状态更新是显式的。不要以副作用方式更新状态，而是连同生成的值一起返回一个新的状态。下面是一个随机数生成器可行的接口：

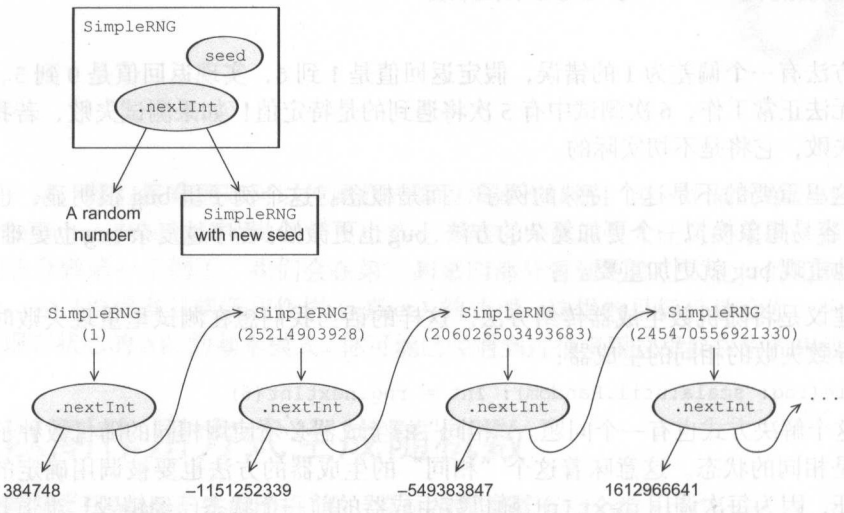
```
trait RNG {
  def nextInt: (Int, RNG)
}
```

这个函数生成一个随机的整数。稍后我们再根据 `nextInt` 定义其他函数。不同于之前在原地更新一些内部状态并只返回一个生成的随机数（同 `Scala.util.Random` 那样），这次返回随机数和一个新的状态，老的状态是不可变的。[3] 实际上我从对新状态和其余程序的通信的关注上分离了对计算下一个状态的关注。没有全局的可变内存被使用——简单地返回下一状态给它的调用者。这给 `nextInt` 的调用者对新状态做什么有完全的控制权。注意，状态依然是被封装的，就这个 API 的使用者而言，并不知道随机数生成器的实现。

3 回忆一下 (A,B) 是两个元素的元组类型，给定 `p: (A, B)` 你可以用 `p._1` 抽取 A，用 `p._2` 抽取 B。

现在我们需要实现这个方法，先挑一种简单的方式，下面是一个简单的与 Scala.util.Random 使用相同算法的随机数生成器。它也被称为“线性同余生成器”（http://mng.bz/r046）。实现的细节并不重要，但要注意 nextInt 在生成下一个随机数时返回了生成的数值以及一个新的 RNG。

一个函数式的随机数生成器（RNG）



每次调用SimpleRNG.nextInt返回序列里的下一个随机数，并且SimpleRNG对象需要对序列延续。

示例 6.2 一个纯函数式随机数生成器

```
case class SimpleRNG(seed: Long) extends RNG {
    def nextInt: (Int, RNG) = {
        val newSeed = (seed * 0x5DEECE66DL + 0xBL) & 0xFFFFFFFFFFFFL
        val nextRNG = SimpleRNG(newSeed)
        val n = (newSeed >>> 16).toInt
        (n, nextRNG)
    }
}
```

&是位与操作，我们用当前种子生成一个新的种子。

下一个状态，用新种子创建的RNG实例。

>>>是右位移操作，高位用0填充。

n是新的伪随机整数。

返回值是一个二元元组，包含随机整数和下一个RNG状态。

下面是在解释器下使用这个 API 的例子：

```
scala> val rng = SimpleRNG(42)
rng: SimpleRNG = SimpleRNG(42)

scala> val (n1, rng2) = rng.nextInt
n1: Int = 16159453
rng2: RNG = SimpleRNG(1059025964525)

scala> val (n2, rng3) = rng2.nextInt
n2: Int = -1281479697
rng3: RNG = SimpleRNG(197491923327988)
```

选择一个任意的种子，42。

声明两个变量，对rng.nextInt返回的二元组解构赋值给这两个变量。

我们可以按照我们想要的多次运行这一系列声明，并总会得到相同的值。当我们调用 rng.nextInt 时它总返回 16159453 和一个新的 RNG，这个新的 RNG 的 nextInt 总是返回 -1281479697，换句话说这个 API 是纯函数式的。

6.3 用纯函数式实现带状态的 API

将带状态的 API 改造为纯函数式风格遇到的问题和解决办法（让 API 计算下一个状态而不改变任何值）不是随机数生成独有的，它是一个频繁发生的问题，我们可以用相同的方式来处理。[4]

举个例子，假设你有这样一个类：

```
class Foo {
  private var s: FooState = ...
  def bar: Bar
  def baz: Int
}
```

假设 bar 和 baz 每次以某种方式改变 s，我们可以通过明确地转换一个状态到下一状态来机械地把它翻译成纯函数式 API：

```
trait Foo {
  def bar: (Bar, Foo)
  def baz: (Int, Foo)
}
```

不论何时使用这一模式，让调用者负责传递计算后的下一状态给剩余的程序。对我们刚展示过的纯函数式的 RNG 接口，如果我们复用一个之前的 RNG，它总会生成一个和之前生成的相同的值。例如：

```
def randomPair(rng: RNG): (Int,Int) = {
  val (i1,_) = rng.nextInt
  val (i2,_) = rng.nextInt
  (i1,i2)
}
```

这里 i1 和 i2 是相同的！如果我们想要生成两个不同的数值，需要使用第一次调用返回的 RNG 来调用 nextInt 生成第二个整数：

```
def randomPair(rng: RNG): ((Int,Int), RNG) = {
  val (i1,rng2) = rng.nextInt
  val (i2,rng3) = rng2.nextInt ←——注意使用rng2。
  ((i1,i2), rng3) ←——在生成两个随机数之后返回最终状态，这让调用者用新状态生成更多随机数。
}
```

4 使用纯函数来计算下一状态会有一些效率的损失，因为我们不能对数据在原地进行修改。（在这儿并不是什么问题，因为需要复制的状态只不过是一个 Long 类型的数据）。这种性能的损失可以通过使用更高效的纯函数式数据结构来缓解。直接对数据在原地进行修改在某些情况下也可能不破坏引用透明，我们会在第四部分展开讨论。

你能看到通用模式，或许你也能看出直接调用 API 很乏味，让我们写一些生成随机数的函数，看看是否可以将重复的地方提取出来。

◈ 练习 6.1

写一个函数使用 RNG.nextInt 生成 0 和 Int.MaxValue 之间（含）的随机数。确保处理极端情况，当 nextInt 返回 Int.MinValue 时不会有一个非负的对应值。

```
def nonNegativeInt(rng: RNG): (Int, RNG)
```

函数式编程中遇到的笨拙实现

当你写更多的函数式程序时，有时会遇到这样一种情况：感觉用函数式表达一段程序很笨拙或乏味。这是否意味着纯函数式就像写一部长篇小说却不使用任何字母 E？当然不是，这种笨拙几乎总意味着缺乏抽象，有待挖掘。

当你遇到这种情况时，我们鼓励你继续向前，并寻找一些可提取的常见模式。大多可能是其他人已经遇到过的，甚至你自己就能重新发现解决方案。即使你被卡在这里，挣扎着苦苦思索寻求方案，这个过程也能帮你更好地理解其他人处理相似问题时已经发现的方案。

多多练习和体验，熟悉这本书包含的一些惯用方案，用函数式来表达程序会变得容易、自然，当然好的设计依然不易，但通过纯函数式编程可以极大地简化设计空间。

◈ 练习 6.2

写一个函数生成 0 到 1 之间的 Double 数，不包含 1。注意，你可以用 Int.MaxValue 获得最大正整数，并使用 x.toDouble 将 Int 转换为 Double。

```
def double(rng: RNG): (Double, RNG)
```

◈ 练习 6.3

写一个函数生成一个 (Int, Double) 对、一个 (Double, Int) 对和一个 (Double, Double, Double) 三元组。应该复用已经写过的函数。

```
def intDouble(rng: RNG): ((Int,Double), RNG)
def doubleInt(rng: RNG): ((Double,Int), RNG)
def double3(rng: RNG): ((Double,Double,Double), RNG)
```

◈ 练习 6.4

写一个函数生成一组随机整数。

```
def ints(count: Int)(rng: RNG): (List[Int], RNG)
```

6.4 状态行为的更好的 API

回顾一下我们的实现，会注意到一个通用模式：我们的每一个函数都有一个形式为 RNG=>(A, RNG) 的类型（存在某种 A 类型）。这种类型的函数被称为*状态行为*（state action）或*状态转移*。因为它们把 RNG 从一个状态转换到另一个状态。状态行为可以通过组合子来组合，组合子是一个高阶函数（会在本章定义）。既然要一直乏味地重复传递状态参数，何不用组合子帮我们在行为之间自动传递状态。

为了让这个行为类型更便于讨论，并简化对它的理解，我们对 RNG 状态行为数据类型定义一个类型别名（type alias）：

```
type Rand[+A] = RNG => (A, RNG)
```

可以把 Rand[A] 类型的值认为是“随机生成 A 类型值的类型”，虽然不是完全精确。它确实是一个状态行为——一段程序依赖某些 RNG 用它来生成一个 A，同时转换 RNG 为一个新的状态可以被另一个行为后续使用。

现在可以把 RNG 的 nextInt 方法变成新类型的一个值：

```
val int: Rand[Int] = _.nextInt
```

我们想写一个组合子让我们能组合 Rand 行为，避免显式地传递 RNG 状态。我们将以某种特定领域语言（DSL）来帮我们实现自动传递。比如，一个简单的 RNG 状态转换是 unit 行为，这个行为只传递 RNG 状态，它总是返回一个常量值（unit）而非随机数。

```
def unit[A](a: A): Rand[A] =
  rng => (a, rng)
```

同样还有 map 也用于转换状态行为的输出而不修改状态自身。记住，Rand[A] 只是函数类型 RNG=>(A, RNG) 的类型别名，所以这只是一种函数的组合：

```
def map[A,B](s: Rand[A])(f: A => B): Rand[B] =
  rng => {
    val (a, rng2) = s(rng)
    (f(a), rng2)
  }
```

看一个 map 如何使用的例子，这个 nonNegativeEven 函数复用 nonNegativeInt 来生成一个大于等于 0 并能被 2 整除的整数：

```
def nonNegativeEven: Rand[Int] =
  map(nonNegativeInt)(i => i - i % 2)
```

◈ **练习 6.5**

使用 map 以更优雅的方式重新实现 double 函数，参见练习 6.2。

6.4.1 组合状态行为

不幸的是 map 还不够强大到可以直接实现练习 6.3 里的 intDouble 和 doubleInt 函数。我们需要一个新的组合子 map2，能够将两个 RNG 行为合并为一个使用二元而非一元的函数。

◈ 练习 6.6

按照下面的签名写一个 map2 函数。这个函数接收两个行为，ra 和 rb，以及一个函数 f 用于组合它们的结果，并返回一个组合了两者的新行为：

```
def map2[A,B,C](ra: Rand[A], rb: Rand[B])(f: (A, B) => C): Rand[C]
```

我们只用实现一次 map2 组合子，然后可以用它对任意 RNG 状态行为进行组合。比如，如果我们有一个生成 A 类型值的行为和一个生成 B 类型值的行为，可以将它们结合成一个生成 (A,B) 对的行为：

```
def both[A,B](ra: Rand[A], rb: Rand[B]): Rand[(A,B)] =
  map2(ra, rb)((_, _))
```

可以用这个重新实现 6.3 练习里的 intDouble 和 doubleInt 函数，将会更简洁：

```
val randIntDouble: Rand[(Int, Double)] =
  both(int, double)

val randDoubleInt: Rand[(Double, Int)] =
  both(double, int)
```

◈ 练习 6.7

难：如果你能组合两个 RNG 转换行为，那么同样也可以组合一个行为列表。实现 sequence 函数将一个转换行为的列表组合为一个转换行为。使用它重新实现之前写过的 ints 函数。对后一种你可以用标准库里的函数 List.fill(n)(x) 创建一个新的列表，并用 n 个重复的 x 填充这个列表。

```
def sequence[A](fs: List[Rand[A]]): Rand[List[A]]
```

6.4.2 嵌套状态行为

一个模式开始浮现：我们正在向着一种不需要显式提及或传递一个 RNG 值的实现方式前进。map 和 map2 组合子允许我们以特别简洁优雅的方式实现，其他实现方式则乏味且容易产生错误。但还有一些函数我们无法根据 map 和 map2 很好实现。

一个这样的函数是 nonNegativeLessThan，生成 0（包含）到 n（不包含）之间的一个整数：

```
def nonNegativeLessThan(n: Int): Rand[Int]
```

第一次尝试实现或许是生成一个非负整数再对 n 求模：

```
def nonNegativeLessThan(n: Int): Rand[Int] =
  map(nonNegativeInt) { _ % n }
```

这当然也会生成一个符合区间范围的数，但有些曲解，因为 Int.MaxValue 或许无法被 n 所除。所以小于（除法的）余数的数字会频繁出现。如果 nonNegativeInt 生成数字大于 32 位中 n 的最大倍数，应该重试生成器，希望得到一个更小的数字。可能会这样尝试：

```
def nonNegativeLessThan(n: Int): Rand[Int] =
    map(nonNegativeInt) { i =>
        val mod = i % n
        if (i + (n-1) - mod >= 0) mod else nonNegativeLessThan(n)(???)
}
```

如果获取数字大于32位中n的最大倍数，重新递归。

这是向右移的方式，但 nonNegativeLessThan(n) 使用了错误的类型。它应该返回 Rand[Int]，接收 RNG 的一个函数。但我们没有，我们希望把行为串起来，这样由 nonNegativeInt 返回的 RNG 可以继续传递给递归调用的 nonNegativeLessThan。我们可以用显式传递的方式代替 map，比如：

```
def nonNegativeLessThan(n: Int): Rand[Int] = { rng =>
    val (i, rng2) = nonNegativeInt(rng)
    val mod = i % n
    if (i + (n-1) - mod >= 0)
        (mod, rng2)
    else nonNegativeLessThan(n)(rng)
}
```

但更好的方式是有一个组合子能帮我们传递，而不是被 map 和 map2 所截掉。我们需要一个更强大的组合子 flatMap。

◈ 练习 6.8

实现 flatMap 然后用它实现 nonNegativeLessThan：

```
def flatMap[A,B](f: Rand[A])(g: A => Rand[B]): Rand[B]
```

flatMap 允许我们通过一个 Rand[A] 来生成一个随机的 A，然后使用这个 A，选择一个 Rand[B]。在 nonNegativeLessThan 里我们根据 nonNegativeInt 生成的值选择是否进行重试。

◈ 练习 6.9

根据 flatMap 重新实现 map 和 map2。这也是 flatMap 比 map 和 map2 更强大的地方。

我们现在可以回顾本章开始的例子，用纯函数式 API 实现一个更加可测的死亡色子？

这里是一个死亡色子 rollDie 的实现，使用 nonNegativeLessThan 依然包含我们之前偏差为 1 的错误。

```
def rollDie: Rand[Int] = nonNegativeLessThan(6)
```

如果我们用各种 RNG 状态测试这个函数，很快就发现 RNG 导致函数返回 0：

```
scala> val zero = rollDie(SimpleRNG(5))._1
zero: Int = 0
```

我们可以用相同的 SimpleRNG(5) 随机数生成器重新实现一个可靠的 rollDie。

修正 bug 是小菜一碟：

```
def rollDie: Rand[Int] = map(nonNegativeLessThan(6))(_ + 1)
```

6.5 更通用的状态行为数据类型

我们写过的函数——unit、map、map2、flatMap 和 sequence 都不是专门为随机数使用的。它们都是处理状态行为的通用函数，不关心状态类型。注意这一点，比如 map 不关心它处理的 RNG 状态行为，我们可以给它一个更通用的签名：

```
def map[S,A,B](a: S => (A,S))(f: A => B): S => (B,S)
```

修改签名并不需要修改 map 的实现。通用签名一直在那儿，只是我们没有注意到。

接着将 Rand 也改为一个通用的签名，可处理任何类型的状态：

```
type State[S,+A] = S => (A,S)
```

这里 State 是对“携带状态的计算”或“状态行为”“状态转换”，甚至是“指令”（statement）的缩写。我们或许想将它定义成自己的类，包装了一个底层的函数，像这样：

```
case class State[S,+A](run: S => (A,S))
```

表达方式并不重要，重要的是我们有了一个单一的、通用的类型。用这个类型可以写出通用目的的函数，捕获状态程序的一般模式。

现在可以把 Rand 作为 State 的别名：

```
type Rand[A] = State[RNG, A]
```

◈ **练习 6.10**

泛化 unit、map、map2、flatMap 和 sequence 函数，把它们尽可能加到 State 类中，否则放在 State 伴生对象中。

这里写过的函数只体现了少数最常见的模式，当你写更多的函数式代码时可能遇到其他模式，并发现其他用于捕获这些模式的函数。

6.6 纯函数式命令编程

在前几节里，我们写的函数遵循一个确定的模式，运行一个状态行为，分配它的结果到一个 val 变量，然后用这个 val 运行另一个状态行为，再把结果分配到另一个 val，等等。看起来很像是命令式（imperative）编程。

在命令式编程范式中，程序是由一系列的指令（statement）组成的，每个指令可以修改程序状态，就像我们之前所做的，只不过我们用的“指令”是一个状态行为（State actions），它是一个真正的函数。作为函数，它们通过接收参数来读取当前程序状态，通过返回一个值来代表写入程序状态。

命令式与函数式编程是对立的吗？

绝对不是。函数式编程是没有副作用的程序，而命令式编程是关于使用指令（statement）修改程序状态的程序，如我们所见，无副作用的维护状态也是完全合理的。

> 函数式编程对写命令式程序有很好的支持，并带来额外的好处。例如程序可以被等式推理，因为它们是引用透明的。我们会在第二部分讨论更多等式推理程序，命令式编程则更多在第三和第四部分讨论。

我们实现了一些如 map、map2 之类的普通组合子，以及 flatMap 终极组合子，来处理状态从一个指令到另一个指令的传播。但这么做看上去失去了一些“命令语气”。

看一下下面的例子（假定我们已经定义了 Rand[A] 是 State[RNG,A] 的类型别名）：

```
val ns: Rand[List[Int]] =
    int.flatMap(x => ←——int是一个类型为Rand[Int]的值，生成一个随机整数。
        int.flatMap(y =>
            ints(x).map(xs => ←——ints(x)生成一个长度为x的list。
                xs.map(_ % y)))) ←——替代list里的每一个元素，用这个元素的值除以y的余数。
```

这里做了什么不容易一下子看出来，不过既然我们已经定义了 map 和 flatMap，可以用 for 推导来还原为命令式风格：

```
val ns: Rand[List[Int]] = for {
    x <- int ←——生成整数x。
    y <- int ←——生成另一个整数y。
    xs <- ints(x) ←——生成一个列表xs，长度为x。
} yield xs.map(_ % y) ←——列表的每个元素对y求模。
```

这段代码更容易读（也容易写），差不多一看便知道大概意图——一段命令式程序维护一些状态。但它与之前的代码实际上是相同的代码。把 nextInt 分配给 x，之后再把下一个 nextInt 分配给 y，然后生成一个长度为 x 的列表，对 y 进行求模，最终返回 map 后的列表。

使用 for 推导（或 flatMap）使命令式编程更容易，只需要两个基本 State 组合子——一个读状态，一个写状态。想象着我们有一个 get 组合子用于获取当前状态，和一个 set 组合子设置当前状态，就可以实现一个以任意方式修改状态的组合子：

```
def modify[S](f: S => S): State[S, Unit] = for {
    s <- get ←——获取当前状态分配给s。
    _ <- set(f(s)) ←——设置新状态s。
} yield ()
```

这个方法返回一个状态行为，用函数 f 修改输入状态，它生成一个 Unit 类型的值，表示除了修改状态没有其他返回值。

get 和 set 行为看起来是怎样的？它们极其简单。get 行为简单地传入输入状态：

```
def get[S]: State[S, S] = State(s => (s, s))
```

set 行为由一个新状态 s 构成。结果行为忽略输入状态，替换为了一个新状态，并返回 () 替代一个有意义的值：

```
def set[S](s: S): State[S, Unit] = State(_ => ((), s))
```

这两个简单的行为与我们写过的状态组合子——unit、map、map2 以及 flatMap 是我们实现函数式风格的任何状态机或带状态程序所需的全部工具。

◇ 练习 6.11

难：为了增加对 State 的使用经验，实现一个对糖果售货机建模的有限状态机。机器有两种输入方式：你可以投入硬币，或可以按动按钮获取糖果。它可以有一种或两种状态：锁定或非锁定。它也可以跟踪还剩多少糖果以及有多少硬币。

```
sealed trait Input
case object Coin extends Input
case object Turn extends Input

case class Machine(locked: Boolean, candies: Int, coins: Int)
```

机器遵循下面的规则：

- 对一个锁定状态的售货机投入一枚硬币，如果有剩余的糖果它将变为非锁定状态。
- 对一个非锁定状态的售货机按下按钮，它将给出糖果并变回锁定状态。
- 对一个锁定状态的售货机按下按钮或对非锁定状态的售货机投入硬币，什么也不做。
- 售货机在“输出”糖果时忽略所有“输入”。

simulateMachine 方法应该根据输入列表返回硬币数和售货机里最终剩余糖果数。比如当前售货机里有 10 个硬币和 5 个糖果，成功地买了 4 个糖果，它的输出应该是 (14,1)。

```
def simulateMachine(inputs: List[Input]): State[Machine, (Int, Int)]
```

6.7 小结

这一章接触的主题是如何写带状态的纯函数式程序。我们用随机数生成作为一个启发性的例子，但它是一个在很多不同领域都存在的一般性问题。基本思想很简单：使用函数式接收一个状态参数，在结果里一并返回一个新的状态。下次你遇到一个依赖副作用的命令式 API 时，看看是否能提供一个纯函数式版本，使用我们这里写过的一些函数会更方便。

第二部分

功能设计和组合子库

第 1 章我们说过，函数式编程是一个基本前提，它将在所有层面上影响着我们如何编写和组织程序。第一部分里，我们介绍了 FP 的基础，也呈现了只使用纯函数如何影响到程序的基本组成部分：循环、数据结构、异常，等等。接下来在第二部分中，我们将会看到函数式编程的假设是如何影响库的设计的。

我们会在第二部分中创建三个有用的库——一个用于并行和异步计算，另一个用于测试程序，第三个用于解析文本。其中不会出现太多新的语法和语言特性，但会使用很多我们在前面章节里涉及过的内容。毕竟我们的首要目标不是教你怎么实现平行计算、测试和解析文本，而是帮助你掌握设计函数式库的技能，哪怕它与前面三者看上去都没什么关系。

本书的这一部分将是一个有点曲折的旅程。在混乱又反复的函数式设计过程中，希望至少能够让你看到基于真实的世界它是如何演进的。不用担心你没能按照讨论的细节来做，这些章节只想让你感觉是从一旁窥视某人的实现过程一样。因为没有两个人会采用完全相同的方式完成这个过程，哪怕在每种情况下的特定路径可能不是让你认为最自然的一种。请记住，当你设计自己的函数式库时，以自己的节奏去做，采取任何你想要的路径，每当面临设计选择时，用任何你认为合适的方式去验证，这可能包括做小实验、创建原型，等等。

在函数式库的设计上没有唯一正确的答案，有的是一堆需要我们取舍的设计选择。我们的目标是让你开始理解这些权衡和不同选择的含义。有时我们会为了教学目的，故意在库的设计中做出一个不良的选择，并在晚些时候才告诉你。那是因为这是在实际的设计中时常会发生的，希望你能体会到这一点。比起第二部分的这些库，更重要的是让你了解函数式设计是怎样的过程，以及如何面对极有可能遇到的情形。库的设计不是少数人才能做的，相反它是普通函数式编程每天都要做的事情。在这些章节中及以后，你绝对应该自由地尝试不同的设计选择，并形成自己的审美。

最后，在第二部分的学习中，你可能会注意到一些重复的代码模式，请先记在你的脑海里。在第三部分，我们将讨论如何消除这种重复，并会发现整个世界的基础抽象对这些库而言是通用的。

7 纯函数式的并行计算

由于现代计算机每个 CPU 都是多核的，且往往有多个 CPU，因此程序设计得能够充分利用它们的并行处理能力就显得尤为重要。然而，并行的程序之间其交互是复杂的，包括以共享内存这样传统的方式实现线程间通信也是出了名的难。这些都将导致程序很容易出现竞争条件（race condition）和死锁（deadlock）的问题，既不好测试，也不利于扩展。

在本章中，我们将构建一个用于创建并行异步计算的库，通过纯函数的描述方式控制并行程序的复杂度。这样一来，我们就可以用代换模型（substitution model）来简化设计的推理过程，并有望让并发计算的工作变得既容易又有趣。

你能从本章学到的不仅仅是如何编写纯函数式的库，还能学到*如何设计一个高度可组合和模块化的库*。为此，我们将始终把*描述*计算本身作为库的设计重点，而非实际*运行*。这样，库的使用者才会从更抽象的层面去编程，而不关心运行的细节。比如到本章的结尾，我们会编写一个组合子 `parMap`，使用它可以轻松对集合的每个元素同时应用函数 `f`：

```
val outputList = parMap(inputList)(f)
```

实现的过程将是循序渐进的。首先，我们挑一个简单的库的使用场景（use case）。然后，设计一个接口契合这个场景。最后，基于这个接口思考如何去实现。随着设计的不断深化，我们在接口与实现之间的来回往复，会让我们对问题（域）的理解更加深入，也让我们的设计能够应对更复杂的场景。另外，我们还会突出*代数推理*（algebraic reasoning）的过程，让大家明白在特定*法则*（law）下，API 是可以用*代数*（algebra）来描述的。

我们为什么要设计自己的库，而不直接使用 `Scala` 标准库中 `scala.concurrent` 包里的 API 呢？我们有两点理由，首先是出于教学的目的，为了让你明白开发这样一个库其实并不难；其次，我们也极力地希望大家意识到，没有哪个现有的库是权威的，或不需要再去推敲的，哪怕它们都是专家们设计的，还冠以“标准”的名头。要知道很多库都存在一些随意的设计选择，尽管大都是无心的。也许依葫芦画瓢是安全的做法，但这并不意味着墨守成规总是最好的选择。一旦从头开始，你就得去重新审视那些已存在于设计库中的基本假设，从不同角度发现一些别人从未想过的问题，也许会让你的设计更好地满足你的目标。好比这里，我们的基本假设就是，库是*绝对不能有副作用*的。

在本章中我们要写很多代码，其中大部分会留给读者做练习。与往常一样，你可以在随书可下载的内容中找到答案。

7.1 选择数据类型和函数

一旦开始设计一个函数式库，它会是什么样的，你总会有一个大致的想法，难得的是完善这些想法，以及找到合适的数据类型满足你想要的函数式性质。就好比“创建并行计算”，可它具体是指什么呢？那我们不妨从一个入手相对容易的并行计算的例子中搞明白——求一组整数的和。下面这个就是用左叠加（left fold）方法计算求和的：

```
def sum(ints: Seq[Int]): Int =
  ints.foldLeft(0)((a, b) => a + b)
```

其中 Seq 是标准库中列表或其他序列的超类。关键是，它有 foldLeft 这个方法。

除了逐一叠加的方法，还可以用一个分治（divide-and-conquer）的算法，请看下面的代码示例：

示例 7.1 通过分治算法求一列整数的和

```
def sum(ints: IndexedSeq[Int]): Int =
  if (ints.size <= 1)
    ints.headOption getOrElse 0
  else {
    val (l,r) = ints.splitAt(ints.length/2)
    sum(l) + sum(r)
  }
```

IndexedSeq是标准库中随机访问序列的超类，如Vector。不像list，它们提供一个有效的splitAt方法基于特定的index将自身一分为二。

headOption是所有集合都定义的方法，第4章见过。

使用splitAt将序列平分。

对两部分数据分别递归求和，再将结果加起来。

我们用 splitAt 函数将序列一分为二，并对各自递归求和，最后合并它们的结果。和基于 foldLeft 实现的方法不同，这种实现是可以并行化的，即对两部分的求和可以同时进行。

简单例子的重要性

在求多个整数的和的计算中，使用并行计算不见得会更快，因为并行计算所带来的开销会多过它能节省的时间。但这样一个简单的例子是可以帮助我们理清设计思路的，反观复杂例子中的细枝末节，只会混淆我们设计的初衷。我们一直期望去阐明问题域本身，而最好的方法就是从简单的例子入手，提取出问题的共同点，再渐进地增加复杂度。在函数式的设计中，构建出一个个简单的可组合的核心数据类型和函数，并用它们灵活地解决问题才是我们的终极目标，而不是某个特定例子。

在思考什么样的数据类型和函数能够将计算并行化时，我们可以变换一种思路。与其绞尽脑汁地用现有的 API（像 java.lang.Thread 和 java.util.concurrent 相关的库）去实现并行计算，不如在上面例子的启发下去设计自己理想的 API，并基于此实现并行计算。

7.1.1 一种用于并行计算的数据类型

如上例中的 sum(l) + sum(r) 分别对两部分进行了 sum 的递归计算。由此看出，用于表示并行计算的任何数据类型都会包含一个结果。这个结果是一个有意义的类型（这里的 Int），且能够被获取到。为此，我们设计这样一个数据类型 Par[A]（Par 是 Parallel 的简写），它像一个装有结果的容器，并且具备下面的方法：

- def unit[A](a: => A): Par[A]，接收一个未求值的 A，返回的结果会在另一个独立的线程中完成求值。我们之所以叫它 unit，是因为它创建了一个并行化单元，其中封装了一个值。
- def get[A](a: Par[A]): A，从并行计算里抽取结果。

就这些真的够吗？是的，当然够！现在，我们不必担心还需要哪些函数，或者 Par 内部的表现是什么，又或者这些函数如何实现。我们只需要从现在的例子中得到必要的数据类型和函数就够了。现在让我们来更新这个例子。

示例 7.2 用自定义数据类型更新求和算法

```
def sum(ints: IndexedSeq[Int]): Int =
    if (ints.size <= 1)
        ints headOption getOrElse 0
    else {
        val (l,r) = ints.splitAt(ints.length/2)
        val sumL: Par[Int] = Par.unit(sum(l))   <——并行计算左半部分。
        val sumR: Par[Int] = Par.unit(sum(r))   <——并行计算右半部分。
        Par.get(sumL) + Par.get(sumR)   <——抽取它们的结果并求和。
    }
```

如上，我们把两个递归的 sum 调用封装到了 unit 的调用当中，然后再用 get 抽取这两个子计算的结果。

直接使用原生并发库的问题

要是用 java.lang.Thread 和 Runnable 会怎样？让我们先看看这些类吧，下面节选了部分 API 并转换为 Scala 代码：

```
trait Runnable { def run: Unit }

class Thread(r: Runnable) {
    def start: Unit   <——开始在一个独立的线程中运行r。
    def join: Unit   <——阻塞当前线程直到r运行结束。
}
```

就此，我们能够从这两个类中看出一个问题，即所有的方法都没有返回有意义的值，那么一旦要从 Runnable 中获取任何信息，则势必会产生副作用（side effect），比如改变一些我们能够检查的状态。这样是不利于组合的，由于要关心 Runnable 的内部行为，导致我们不能以更通用的方式操作它。Thread 还有另

外的缺点，就是它直接对应操作系统的线程，是太过稀缺的资源。相对于我们的问题而言，最好是能创建很多的“逻辑线程”，至于它们如何映射到操作系统的线程我们后面会谈到。

不是还有 `java.util.concurrent.Future` 和 `ExecutorService` 吗？为什么不用这些呢？我们接着看：

```
class ExecutorService {
    def submit[A](a: Callable[A]): Future[A]
}
trait Future[A] {
    def get: A
}
```

尽管它们在很大程度上对物理线程进行了抽象，可相对于我们所期望的程度仍然处于底层。比方说，一旦调用 `Future.get`，这将阻塞当前线程直到 `ExecutorService` 执行完，而这样的 API 设计对于组合多个 `Future` 是毫无意义的。当然，我们可以基于这些工具类实现自己的类库（事实上也是这么做的），但它们却不是那种模块化的可组合的API，能够让我们在函数式程序中直接去使用。

现在我们正面临一个选择，是让 `unit` 在一个独立的（逻辑）线程[1]中立即求值呢，还是等到 `get` 被调用时再求值呢？仔细看代码，要想获得任何程度的平行，这就要求两个 `unit` 同时对参数求值并立即返回。你能看出原因吗？[2]

但是这样做了可以说是破坏了引用的透明性。我们不妨将 `sumL` 和 `sumR` 替换成它们的定义，原则上是不影响计算结果的，可这样的程序是无法并行的：

```
Par.get(Par.unit(sum(l))) + Par.get(Par.unit(sum(r)))
```

如上，一旦 `unit` 对参数进行求值后，`get` 就会等待求值的结果。换言之，内联了 `sumL` 和 `sumR` 两个变量之后，`+` 两边的表达式是无法实现平行执行的。由此可见，`unit` 存在着一定的副作用，且仅与 `get` 相关。你看，即便是 `unit` 返回一个代表着异步计算的 `Par[Int]`，可就在它被传递给 `get` 时，显性地去等待计算的结果就暴露了其中的副作用。这么一来就让我们不得不去避免调用 `get`，或者说是直到最后才去调用它，从而实现在无须等待它们完成的情况下，整合这些异步计算。

在继续往下之前，让我们总结一下。首先，我们编出了一个简单的、几乎微不足道的例子。在稍微的探讨后提出了一个设计上的选择。然后，通过一些实验，在一个选项中发现了有趣的结果，并从中学到了一些关于问题域内在的基本事实！整个设计过程就如同一个个探险的过程，这既不需要你有执照，也不用你是函数编程的专家，就这么走起，看看你能发现什么吧！

---

1 本章使用的*逻辑线程*这个术语表示程序中与主线程同时运行着的计算，它并非与操作系统的线程一一对应。具体来讲，是通过线程池的方式，将大量逻辑线程映射到少量的操作系统线程上。

2 在 Scala 中函数参数是严格按照从左至右的顺序求值的，那么只要 `unit` 是在 `get` 被调用时才延迟执行的，我们将同时衍生（spawn）出并行计算，并在衍生下一个并行计算之前等待前一个并行计算完成，这样的计算比较高效同时也保证串行执行。

7.1.2 组合并行计算

我们来试试看能否避免整合 unit 和 get 的缺陷。如果我们不调用 get，这同时意味着函数 sum 必须返回一个 Par[Int]。这样的变化会呈现什么结果呢？让我们先根据函数签名实现了再说：

```
def sum(ints: IndexedSeq[Int]): Par[Int] =
    if (ints.size <= 1)
        Par.unit(ints.headOption getOrElse 0)
    else {
        val (l,r) = ints.splitAt(ints.length/2)
        Par.map2(sum(l), sum(r))(_ + _)
    }
```

◈ **练习 7.1**

Par.map2 是一种新的高阶函数，用于组合两个并行计算的结果。它的签名会是怎样的呢？请给出最为通用的签名描述（不要局限于 Int）。

如你所见我们没有在递归中调用 unit，尽管目前尚不清楚是否应该让 unit 来接收延迟的参数，但至少在本例中，接收延迟的参数没有带来任何好处。也许这并非总是如此，不妨让我们稍后再回到这个问题。

那么 map2 应该接收延迟的参数吗？这貌似是合理的，map2 让两边的计算得到均等的平行计算机会（就参数顺序而言这样的做法看似有点武断，但现在我们只想要 map2 实现二者计算合并是独立的，且可以并行运行）。这样的选择真能实现我们想要的吗？我们不妨以 sum(IndexedSeq(1,2,3,4)) 作为测试案例，看看 map2 在严格（strict）求值的过程中会发生什么。这里我们花上一点时间跟随着程序的运算轨迹（trace）推演和理解整个过程。

示例 7.3 求和的程序轨迹

```
sum(IndexedSeq(1,2,3,4))
map2(
    sum(IndexedSeq(1,2)),
    sum(IndexedSeq(3,4)))(_ + _)
map2(
    map2(
        sum(IndexedSeq(1)),
        sum(IndexedSeq(2)))(_ + _),
    sum(IndexedSeq(3,4)))(_ + _)
map2(
    map2(
        unit(1),
        unit(2))(_ + _),
    sum(IndexedSeq(3,4)))(_ + _)
map2(
    map2(
        unit(1),
```

```
        unit(2))(_ + _),
    map2(
        sum(IndexedSeq(3)),
        sum(IndexedSeq(4)))(_ + _))(_ + _)
...
```

如上所示的轨迹，正如之前章节里做的一样，我们用 sum 的定义替换 sum 本身。map2 在先左后右的顺序下对参数递归求值，即 sum(IndexedSeq(1,2)) 将比 sum(IndexedSeq(3,4)) 先展开求值。即便 map2 是并行（任何可实现的方式，比如线程池）对参数求值，也是先左后右地进行。

如果 map2 在保持先左后右不变的同时，但并不马上进行求值，这会有帮助吗？假定如此，那么 Par 仅是用来构建一个并行计算的*描述*，实际上什么也不会发生，直到对这个描述真正*求值*，类似调用 get 函数一样。问题是，如果只是构建一个描述，它将是一个重量级（heavyweight）对象，包含了所有将要执行的操作树：

```
map2(
    map2(
        unit(1),
        unit(2))(_ + _),
    map2(
        unit(3),
        unit(4))(_ + _))(_ + _)
```

无论使用什么样的数据结构去存储这样的描述，都将占用比列表本身更多的空间。要是能让我们的描述更轻量级（lightweight）一些该多好啊。

看来只能让 map2 是惰性（lazy）的，才能让左右两部分并行地立即执行起来，也解决了先后的问题。

7.1.3 显性分流

即便选定了，可还是觉得哪里不对劲。难道所有情况下我们都想并行地求值左右两部分吗？我想不太可能，请看下面这个例子：

```
Par.map2(Par.unit(1), Par.unit(1))(_ + _)
```

很明显，这种情况下直接执行会更快一些，我们并不希望异步到独立的逻辑线程中执行。可 API 无法让我们做出不同的选择。也就是说，目前的 API *没有明确地*表明何时应该将计算从主线程分流出去（forked off），换句话说程序员也不确定哪儿会发生并行计算。如果我们让分流（forking）更明确呢？我们可以通过引入另一个函数来做到，def fork[A](a: => Par[A]): Par[A]，它可以将 Par 分配到另一个独立的逻辑线程中运行：

```
def sum(ints: IndexedSeq[Int]): Par[Int] =
    if (ints.length <= 1)
        Par.unit(ints.headOption getOrElse 0)
    else {
        val (l,r) = ints.splitAt(ints.length/2)
        Par.map2(Par.fork(sum(l)), Par.fork(sum(r)))(_ + _)
    }
```

现在我们可以让 map2 保持不变的同时，让程序员使用 fork 来决定何时要并行。fork 函数不仅解决并行计算过于严格（strictly）的问题，更是从根本上让并行编程可控制了。这里其实我们解决了两个问题，第一个是，我们需要一些方法来表明这两条并行任务的结果应该合并；另一个则是，是否选择让一个特定的任务异步执行。通过将它们分开讨论，让我们避免将一些全局的并行策略应用到 map2 及其他函数上。

现在让我们回到 unit 是应该严格（strict）还是惰性（lazy）的问题上来。有了 fork，即便 unit 是严格（strict）的也不会有任何损失。至于非严格的版本，让我们叫它 lazyUnit 吧，可使用 unit 和 fork 组合实现：

```
def unit[A](a: A): Par[A]
def lazyUnit[A](a: => A): Par[A] = fork(unit(a))
```

函数 lazyUnit 就是派生组合子的一个简单的例子，它与原生的组合子 unit 恰好相反。我们可以定义 lazyUnit 这样的操作，但在后文中，我们最终决定 Par 是什么样子的时候，是不会有 lazyUnit 的，而只会有 fork 和 unit。[3]

尽管我们知道 fork 是要让参数到另一个逻辑线程中去求值，可我们是该让它在被调用时立即触发求值呢，还是等到类似 get 调用时再触发求值呢？换句话说，求值到底是 fork 的职责还是 get 的职责，或者说是应该当即执行还是延后执行？面临这样不确定的选择时，我们总是要继续下去，直到后面某个点能够清楚地权衡不同选择的利弊。这里有个实用的诀窍，即这样的职责对于 fork 或 get 的实现而言，谁更不可或缺，就是谁的职责。

如果在 fork 里立即并行对参数求值，则实现必须清楚地知道如何创建线程和提交任务到线程池，无论直接与否。更进一步地说，这意味着我们调用 fork 的地方，线程池（或其他任何实现并行的资源）必须是可（全局）访问的且被正确初始化的。[4] 为此，我们将失去在程序其他地方控制并行策略的能力。虽然全局做法没有什么不对，但我们可以想象一下，在别的地方也能控制会更好（比如，一个大型应用的子系统可以通过不同参数来定制自己的线程池）。这样看来我们将创建线程和提交任务的职责放在 get 里更合适了。

发现了没，做出这样的结论不需要你很清楚如何实现 fork 和 get，或是 Par 最终被定义成什么样。我们只是推理一下衍生并行任务的必要信息，探究了 Par 知道了这些信息的后果而已。

相反，如果 fork 只是简单地持有未求值的参数到最后，它就无须关心并行实现的机制，仅是给未求值的 Par 打了一个要并行的“标记”而已。有了这个模型，Par 本身并不需要知道如何真正实现并行。它更多的是并行计算的描述，待晚些时候由像 get 之类的函数来解释。在此之前，Par 只是一个值的容器，在可用的时候以便我们获取其值。现在则更多的是我们可以运行一等对象（first-class program）。因此，让我们重命名 get 函数为 run，以表明这是并行实际执行的地方：

---

3 这意味着函数应该更为通用和抽象，不仅仅是对 Par 而言，任何类型也是如此。关于这一点我们还会在本书的第三部分深入讨论。

4 第 1 章咖啡的例子中，信用卡处理系统是需要可访问 buyCoffee 方法。

```
def run[A](a: Par[A]): A
```

因为 Par 现在还是一个纯粹的数据结构，run 则赋予了它实现并行的含义，是否要创建线程，将任务委派给线程池或其他什么类似的机制。

7.2 确定表现形式

经过各种思考和选择后，我们有了下面大致的 API。

示例 7.4 Par 的 API 草稿

```
def unit[A](a: A): Par[A]                                    ←——创建一个即时结果为a的计算。
def map2[A,B,C](a: Par[A], b: Par[B])(f: (A,B) => C): Par[C] ←——通过一个二元函数合并两个并行计算为一个新的并行计算。
def fork[A](a: => Par[A]): Par[A]                            ←——将计算标记为在run时进行并发求值。
def lazyUnit[A](a: => A): Par[A] = fork(unit(a))             ←——将表达式包装为run时并发求值计算。
def run[A](a: Par[A]): A                                     ←——对fork标记的Par进行并发的求值，并最终返回结果。
```

这些函数的含义分别是：

- unit 将一个恒定值变为一个并行计算。
- map2 通过一个二元函数合并两个并行计算为一个新的并行计算。
- fork 代表要并发的计算，实际上不会进行求值，直到 run 被调用。
- lazyUnit 包装一个标记为并发的未求值计算。
- run 从实际执行的计算中获取结果值。

在勾画出一个 API 的同时，随时可以开始考虑出现的抽象类型实现的可能性。

◇ **练习 7.2**

在继续之前，尽可能实现 API 中的函数。

要怎么动手呢，我们知道 run 需要以某种方式执行异步任务，与其写我们自己的低级 API，倒不如使用 Java 标准库中的 java.util.concurrent.ExecutorService。下面是它的 Scala 代码：

```
class ExecutorService {
    def submit[A](a: Callable[A]): Future[A]
}
trait Callable[A] { def call: A }  ←——实质上就是惰性类型A。
trait Future[A] {
    def get: A
    def get(timeout: Long, unit: TimeUnit): A
    def cancel(evenIfRunning: Boolean): Boolean
    def isDone: Boolean
    def isCancelled: Boolean
}
```

ExecutorService 让我们提交一个 Callable 的参数（在 Scala 中我更愿意提交一个惰性参数）并返回相应的 Future，一个运行在另一个线程的计算句柄。Future 的 get 方法可以获取结果值（这会阻塞线程直到值被算出来），它还有用于取消执行的额外性质（在阻塞一定时候后抛出异常之类）。

让我们试着假设 run 函数可以访问一个 ExecutorService，看看是否能搞清楚 Par 的样子：

```
def run[A](s: ExecutorService)(a: Par[A]): A
```

最简单的样子莫过于，Par[A] 是 ExecutorService => A，当然，这未免让 run 太过于简单了。要是能让调用者自定义一个超时，或可以取消那就更好了。为此 Par[A] 应该是 ExecutorService =>Future[A]，而 run 就直接返回 Future：

```
type Par[A] = ExecutorService => Future[A]

def run[A](s: ExecutorService)(a: Par[A]): Future[A] = a(s)
```

需要注意的是，Par 是一个以 ExecutorServcie 作为输入的函数，只有在 ExecutorService 被提供后，Future 才会被创建。

真的就这么简单吗？先这样吧，等到它不能满足对函数式的要求时再来修正它。

7.3 完善 API

到目前为止，整个过程还是有那么一点不真实。在实践当中，API 的设计与表现形式的选择是没有明确界限的，更没有先后关系。表现形式的想法可以启发 API，同样 API 也可以促进表现形式的选择，二者之间自然流转，这就像疑问产生时，我们会去实验，构建原型，然后产生新的疑问，周而复始，迭代进化。

这一节将围绕 API 展开讨论。我们在这个简单的例子中已经收获了很多，以至于我们可以尝试添加一些新的基本操作，但在此之前还是有必要将我们学到的内容消化得更透一些。我们已经基于这些已有的基本操作和含义，勾勒出了问题域的范围，现在是时候去完善它们啦。这将会是也应该是一个流畅的过程，任何时候我们都可以改变问题域的规则，甚至是对表现形式的根本改变或添加一个新的基本操作，与此同时还要搞清楚它们是怎样发生的。

让我们从实现 API 开始动手吧。既然有了 Par 的表现类型，不妨就简单直接一点。下面就是基于 Par 的表现类型最简单的实现。

示例 7.5 Par 的基本实现

```
object Par {
    def unit[A](a: A): Par[A] = (es: ExecutorService) => UnitFuture(a)
    unit表现为一个返回UnitFuture的函数，而UnitFuture就是包装了一个不变量的Future的简单实现。
    unit并不调用ExecutorService，也就不能取消，只是简单地在get调用时返回其值。

    private case class UnitFuture[A](get: A) extends Future[A] {
        def isDone = true
```

```
    def get(timeout: Long, units: TimeUnit) = get
    def isCancelled = false
    def cancel(evenIfRunning: Boolean): Boolean = false
  }

  def map2[A,B,C](a: Par[A], b: Par[B])(f: (A,B) => C): Par[C] =
    (es: ExecutorService) => {
      val af = a(es)
      val bf = b(es)
      UnitFuture(f(af.get, bf.get))
    }

  def fork[A](a: => Par[A]): Par[A] =
    es => es.submit(new Callable[A] {
      def call = a(es).get
    })
}
```

map2并没有在另一个线程中调用f，而是由fork来并行控制。只要你愿意，总是可以fork(map2(a, b)(f))在另一线程中执行。

map2的实现并不考虑超时的问题。它仅是简单将ExecutorService传递给两个Par，并等待af和bf的结果，再应用f，最终包装成一个UnitFuture。要是考虑超时的话，我们需要一个新的Future可以记录af的耗时，并减去bf的耗时。

这是最简单自然的实现，可它是存在问题的——外部的Callable将会阻塞直到内部的任务完成。阻塞会导致线程池的线程被占用，这就意味着并行度会因此而降低。本质上讲，一个线程就能够完成的事情，我们用了两个线程。这种实现的严重问题我们将在本章后面讨论。

我们应该注意到了，Future 是没有一个纯粹的功能接口的。这就是我们不希望库的用户直接使用 Future 的原因。但重要的是，即使 Future 的方法是有副作用的，可整个 Par API 仍然是无副作用的。只有在用户调用 run 并且传入 ExecutorService 时，Future 才会暴露出来。简单来讲，接口虽纯（pure），但实现最终还是不纯的。其中的细节我们留待第四部分中讨论。

◇ 练习 7.3

难：改进 map2 的实现以支持超时设置。

◈ 练习 7.4

使用 lazyUnit 写一个函数将另一个函数 A => B 转换为一个异步计算。

```
def asyncF[A,B](f: A => B): A => Par[B]
```

使用隐式转换添加中缀（infix）语法

如果 Par 是一个实际的数据类型，那么像 map2 这样的函数可以被作为 Par 的方法，以实现 x.map2(y)(f) 这样的中缀语法（就像曾为 Stream 和 Option 做的那样）。但是 Par 现在只是一个类型别名，无法直接做到这一点。若是使用隐式转换是可以做到的。这里我们就不展开讨论了，毕竟是题外话，有兴趣的话可以从本章附带的源码中找到答案。

用现有的组合子我们还可以表达什么呢？让我们看一个更具体的例子。

ExecutorService 让我们提交一个 Callable 的参数（在 Scala 中我更愿意提交一个惰性参数）并返回相应的 Future，一个运行在另一个线程的计算句柄。Future 的 get 方法可以获取结果值（这会阻塞线程直到值被算出来），它还有用于取消执行的额外性质（在阻塞一定时候后抛出异常之类）。

让我们试着假设 run 函数可以访问一个 ExecutorService，看看是否能搞清楚 Par 的样子：

```
def run[A](s: ExecutorService)(a: Par[A]): A
```

最简单的样子莫过于，Par[A] 是 ExecutorService => A，当然，这未免让 run 太过于简单了。要是能让调用者自定义一个超时，或可以取消那就更好了。为此 Par[A] 应该是 ExecutorService =>Future[A]，而 run 就直接返回 Future：

```
type Par[A] = ExecutorService => Future[A]

def run[A](s: ExecutorService)(a: Par[A]): Future[A] = a(s)
```

需要注意的是，Par 是一个以 ExecutorServcie 作为输入的函数，只有在 ExecutorService 被提供后，Future 才会被创建。

真的就这么简单吗？先这样吧，等到它不能满足对函数式的要求时再来修正它。

7.3 完善 API

到目前为止，整个过程还是有那么一点不真实。在实践当中，API 的设计与表现形式的选择是没有明确界限的，更没有先后关系。表现形式的想法可以启发 API，同样 API 也可以促进表现形式的选择，二者之间自然流转，这就像疑问产生时，我们会去实验，构建原型，然后产生新的疑问，周而复始，迭代进化。

这一节将围绕 API 展开讨论。我们在这个简单的例子中已经收获了很多，以至于我们可以尝试添加一些新的基本操作，但在此之前还是有必要将我们学到的内容消化得更透一些。我们已经基于这些已有的基本操作和含义，勾勒出了问题域的范围，现在是时候去完善它们啦。这将会是也应该是一个流畅的过程，任何时候我们都可以改变问题域的规则，甚至是对表现形式的根本改变或添加一个新的基本操作，与此同时还要搞清楚它们是怎样发生的。

让我们从实现 API 开始动手吧。既然有了 Par 的表现类型，不妨就简单直接一点。下面就是基于 Par 的表现类型最简单的实现。

示例 7.5 Par 的基本实现

```
object Par {
  def unit[A](a: A): Par[A] = (es: ExecutorService) => UnitFuture(a)
  unit表现为一个返回UnitFuture的函数，而UnitFuture就是包装了一个不变量的Future的简单实现。
  unit并不调用ExecutorService，也就不能取消，只是简单地在get调用时返回其值。

  private case class UnitFuture[A](get: A) extends Future[A] {
    def isDone = true
```

```
    def get(timeout: Long, units: TimeUnit) = get
    def isCancelled = false
    def cancel(evenIfRunning: Boolean): Boolean = false
  }

  def map2[A,B,C](a: Par[A], b: Par[B])(f: (A,B) => C): Par[C] =
    (es: ExecutorService) => {
      val af = a(es)
      val bf = b(es)
      UnitFuture(f(af.get, bf.get))
    }

  def fork[A](a: => Par[A]): Par[A] =
    es => es.submit(new Callable[A] {
      def call = a(es).get
  })
}
```

map2并没有在另一个线程中调用f，而是由fork来并行控制。只要你愿意，总是可以fork(map2(a, b)(f))在另一线程中执行。

map2的实现并不考虑超时的问题。它仅是简单将ExecutorService传递给两个Par，并等待af和bf的结果，再应用f，最终包装成一个UnitFuture。要是考虑超时的话，我们需要一个新的Future可以记录af的耗时，并减去bf的耗时。

这是最简单自然的实现，可它是存在问题的——外部的Callable将会阻塞直到内部的任务完成。阻塞会导致线程池的线程被占用，这就意味着并行度会因此而降低。本质上讲，一个线程就能够完成的事情，我们用了两个线程。这种实现的严重问题我们将在本章后面讨论。

我们应该注意到了，Future 是没有一个纯粹的功能接口的。这就是我们不希望库的用户直接使用 Future 的原因。但重要的是，即使 Future 的方法是有副作用的，可整个 Par API 仍然是无副作用的。只有在用户调用 run 并且传入 ExecutorService 时，Future 才会暴露出来。简单来讲，接口虽纯（pure），但实现最终还是不纯的。其中的细节我们留待第四部分中讨论。

◇ 练习 7.3

难：改进 map2 的实现以支持超时设置。

◈ 练习 7.4

使用 lazyUnit 写一个函数将另一个函数 A => B 转换为一个异步计算。

```
def asyncF[A,B](f: A => B): A => Par[B]
```

使用隐式转换添加中缀（infix）语法

如果 Par 是一个实际的数据类型，那么像 map2 这样的函数可以被作为 Par 的方法，以实现 x.map2(y)(f) 这样的中缀语法（就像曾为 Stream 和 Option 做的那样）。但是 Par 现在只是一个类型别名，无法直接做到这一点。若是使用隐式转换是可以做到的。这里我们就不展开讨论了，毕竟是题外话，有兴趣的话可以从本章附带的源码中找到答案。

用现有的组合子我们还可以表达什么呢？让我们看一个更具体的例子。

假设有一个 Par[List[Int]] 代表一个产出 List[Int] 的并行计算，比如说排序：

```
def sortPar(parList: Par[List[Int]]): Par[List[Int]]
```

当然，我们可以调用 run，对列表排序，然后用 unit 包装成一个 Par。然而，要实现避免调用 run，我们唯一可以用来操作值的组合子只有 map2。因此，如果把 parList 作为 map2 的一个参数，我们就可以对 List 进行排序了，而另一个参数随便给什么都行，比如说空操作：

```
def sortPar(parList: Par[List[Int]]): Par[List[Int]] =
  map2(parList, unit(()))((a, _) => a.sorted)
```

我们很容易地就实现了对 Par[List[Int]] 进行排序。接下来，我们不妨进一步泛化它，我们可以“lift”任何 A => B 的函数将一个 Par[A] 变成一个 Par[B]，即用任何函数 map 一个 Par：

```
def map[A,B](pa: Par[A])(f: A => B): Par[B] =
  map2(pa, unit(()))((a,_) => f(a))
```

那么，sortPar 现在简单写成：

```
def sortPar(parList: Par[List[Int]]) = map(parList)(_.sorted)
```

看上去简单明了。这就是通过组合基本的操作让类型串联起来的效果，你应该能从用 map2 和 unit 实现 map 中看出来这一点。

传入一个参数，然后忽略它，这算是一种欺骗的手段吗？我不这么认为，事实上我们可以参照 map2 的方式实现 map，而不是其他的方式，就是表明 map2 确实比 map 强大。这样的情况在设计库时经常发生，看似是一个基本函数其实是由一些更强大的基本函数实现的。

还可以用我们的 API 实现什么呢？能否将整个列表都并行化呢？不像 map2 只是组合两个并行计算，parMap（暂且这么叫）可以组合 *n* 个并行计算。它看起来应该是这样的：

```
def parMap[A,B](ps: List[A])(f: A => B): Par[List[B]]
```

我们总是可以将 parMap 作为一个基本函数来声明，不要忘了 Par[A] 只是 ExecutorService => Future[A] 的别名而已。

将任何操作作为基本函数来实现没有什么不对的。可有些情况下，将其作为某个特定类型的方法实现也许会更好。但是现在我们感兴趣的是，哪些操作是可以用已有的 API 表达的，以及掌握我们所定义的各种操作之间的关系。理解什么样的组合子是真正基础的将第三部分变得更加重要，届时我们将展现如何从库中抽取公共的模式。[5]

让我们看看仅用已有的组合子能够把 parMap 实现到什么程度：

```
def parMap[A,B](ps: List[A])(f: A => B): Par[List[B]] = {
  val fbs: List[Par[B]] = ps.map(asyncF(f))
  ...
}
```

5 本例中，有另外一个理由不把 parMap 作为一个基础函数来实现——要想正确地实现是很有挑战的，尤其是恰当地引入对超时的考虑。这种情况常常出现，一个基础的组合子封装了某些特别的逻辑，重用它意味着我们不想重复这个逻辑。

还记得吗，asyncF 是通过 fork 实现 A => B 到 A => Par[B] 的转换的。因此，我们可以很容易将列表转换成 *n* 个并行计算，但是我们需要一种方法收集它们的结果。难道我们就卡在这儿了吗？嗯，仅从 parMap 的函数签名上可以看出，我们需要一个能将 List[Par[B]] 转换为 Par[List[B]] 的方法。

◈ **练习 7.5**

难：实现一个叫 sequence 的函数。不能使用额外的基础函数，不能调用 run。

```
def sequence[A](ps: List[Par[A]]): Par[List[A]]
```

一旦有了 sequence 函数，我们便可以实现 parMap 了：

```
def parMap[A,B](ps: List[A])(f: A => B): Par[List[B]] = fork {
  val fbs: List[Par[B]] = ps.map(asyncF(f))
  sequence(fbs)
}
```

注意，我们已经在实现中调用了 fork，因此无论输入的列表有多大 parMap 都将立即返回。当我们稍后调用 run 时，fork 将派生 *n* 个并行计算，然后等待这些计算到结束，收集它们的结果并返回。

◈ **练习 7.6**

实现 parFilter，并行过滤列表元素。

```
def parFilter[A](as: List[A])(f: A => Boolean): Par[List[A]]
```

你有能想到的任何其他有用的函数要写吗？试着自己写些并行计算，看看哪些可以无须额外的基本函数。这里就有一些想法，你可以尝试：

- 是否可以将章节之初的函数变得更通用？尝试使用它并行来找到 IndexedSeq 的最大值。
- 写一个函数，并行计算一组段落（List[String]）并返回所有段落字的总数。尽可能地泛化这个函数。
- 参照 map2 实现 map3、map4、map5。

7.4 API 与代数

正如前面演示的，我们经常只是写下类型签名，然后“按照类型”来实现。以这种方式工作，我们几乎不关心具体的细节（比如当 map 的实现是组合 map2 和 unit 时），只专注于类型的串联。这并非欺诈，而是自然的推导风格，类似于我们简化代数方程时做的

推导。把 API 看作是代数（algebra），[6] 或是一系列法则或属性的抽象操作，然后基于这样的代数游戏规则进行符号操纵。

到现在为止，我们已经在 API 上做过一些非正式的推导了。这样做不仅没有什么错，还能帮你抽象那些你期望（或愿意）包含在 API 中的法则。[7] 如果没有意识到这一点，那这些法则将只会停留在你的脑海里。实际地将它们写下来并精确标注，可以凸显你设计的选择，而不至于让推导流于形式。

7.4.1 映射法则

如同任何设计选择一样，法则的选择会限定什么操作有什么样的含义，决定什么实现选择更可行，还将影响怎样的性质会为真。让我们来看一个例子。虚构一个看起来合理的法则，在测试的时候它可以被用来测试我们编写的库：

```
map(unit(1))(_ + 1) == unit(2)
```

函数 `_ + 1` 将 `unit(1)` 映射为了 `unit(2)`（法则常常是从标识的具体例子[8] 中得出来的）。在某种意义上它们是等价的？这是一个有趣的问题。为此，我们可以说对于任何有效的 `ExecutorService` 而言，两个相同的 `Par` 对象，其 `Future` 获取的结果值都是一样的。

我们可以验证这一点是否适用于特定的 `ExecutorService` 的函数：

```
def equal[A](e: ExecutorService)(p: Par[A], p2: Par[A]): Boolean =
  p(e).get == p2(e).get
```

法则和函数有很多共同点。正如我们可以概括函数，我们也可以概括法则。例如，前面的函数可以泛化为：

```
map(unit(x))(f) == unit(f(x))
```

在这里我们可以用 `x` 替代 `1`，用 `f` 替代 `_ + 1`。同时隐含实现上的约束。比方说，`unit` 的实现就不能将输入的 `1`，经过计算后返回 `42` 的结果。对于 `ExecutorService` 也是一样，对于我们提交的 `Callable` 对象不能有任何假设或基于接收参数的值来改变行为。更具体地说，这一法则不允许 `map` 和 `unit` 的实现中出现向下转型或 `isInstanceOf` 检查（通常伴随着类型转换）。

就像我们定义简单的函数只做一件事情一样，简单的法则也可以被定义成只规定一件事情。让我们看看是否能够进一步简化这一法则。我们说过，我们希望这个法则对任何 `x`

---

6 这里指的代数不仅仅是数学意义上的，还包括一组对象上的操作，以及其他公理。公理是那些我们假定为真，且能从其他定理推导为真的陈述。具体到我们的例子中就是，一些类型像 `Par[A]` 和 `List[Par[A]]` 的系列，其应用的函数也会像 `map2`、`unit` 和 `sequence`。

7 关于法则我们会在本书余下的部分谈得更多。在下一章中，我们会设计一个声明式的测试库，它允许我们定义函数应该满足的性质，并自动生成测试用例来验证。在第三部分，我们将引入一些法则来规定抽象接口。

8 这里的标识我们想表达的是，在数学上指两个表达式相同或等价。

和 f 都有效，一旦我们用等价的函数[9]替换掉 f，就会出现一些有趣的事情。对等式两边同时简化，我们可以得到一个新的更简单的法则：[10]

```
map(unit(x))(f) == unit(f(x))  <——初始法则。
map(unit(x))(id) == unit(id(x))  <——用id替换f。
map(unit(x))(id) == unit(x)  <——简化。
map(y)(id) == y  <——两边都用y替换unit(x)。
```

醉了吧！从最后的法则可以看出对于 map 而言 unit 明显是个多余的细节。为了进一步弄明白新的法则，让我们想想 map 不能做什么。比如说，它不能抛出异常，不能在应用函数之前崩溃（你看出来为什么这违反了法则了吗？）。所有它能做的就是应用函数 f 去得到结果 y，而当函数是 id 时则对 y 不做任何变化就返回了。[11]更有趣的是，给定 map(y)(id) == y，通过反向替换我们可以得到最初的法则，那个更复杂的法则。（试试看！）从逻辑上讲，我们这样做没有任何限制，因为 map 不可能在不同的函数类型时表现出不同的行为。因此，给定 map(y)(id) == y，那么 map(unit(x))(f) == unit(f(x)) 一定是成立的。由于我们得到的这个第二法则或定理是自由的，仅仅是因为 map 的参数态（parametricity），它有时也被称为*自由定理*（free theorem）。[12]

◇ **练习 7.7**

难：给定 map(y)(id) == y，可由自由定理得到 map(map(y)(g))(f) == map(y)(f compose g)（有时这被称为 *map 融合*，并且它可以作为一种优化，而不是生成一个单独的并行计算来计算第二映射，我们可以把它折叠成第一映射）。[13]你能证明这一点？你可能需要阅读论文“Theorems for Free!”（http://mng.bz/Z9f1），以更好地理解自由定理的“诀窍”。

7.4.2 分流法则

这里面有趣的是，法则并没有约束我们的实现。你可能在无意识的状态下就假定了这些性质（毕竟 map、unit 或 ExecutorService.submit 的实现上出现特殊情况，或是 unit 随机地抛出异常，都会显得很奇怪）。让我们考虑一条更强的性质，即 fork 不应该影响一个并行计算的结果：

```
fork(x) == x
```

---

9 这个等价的函数是：def id[A](a: A): A = a。

10 这和我们在代数方程式里做的简化过程一样。

11 这里的 map 指的是必须保存结构的，即它不会改变并行计算的结构，除了计算“内部”的值。

12 “自由定理”的观点来自于 Philip Wadler 的经典论文《Theorem for Free》（http://mng.bz/Z9f1）。

13 Par 现在的表现类型不能进行这样的优化，因为它是不透明的函数。如果它被物化为一种数据类型，我们可以通过模式匹配和机会发现来应用此规则。你不妨试验一下。

这条性质的成立貌似是显而易见的、最理想的，甚至和我们期望中是一致的。`fork(x)` 应该对 `x` 做同样的事情，只不过是在主线程分离的逻辑线程中异步执行罢了。如果这项法则并不总是成立，我们就必须以某种方式知道什么时候才是不改变其含义的安全的调用，即便是没有类型系统的帮助。

出人意料的是，这个简单的性质对 `fork` 的实现形成了很强的约束。写下这样的法则后，脱下你实现者的帽子，戴上你调试者的帽子，并尝试打破你的法则。思考任何可能的低概率情况，试着举出法则有效的反例，哪怕建立一个非正式的证据，只要足以说服持怀疑态度的程序员。

7.4.3 打破法则：一个微妙的 bug

尝试一下这种思维模式。我们一直认为 `fork(x) == x` 对任意 `x` 和任意 `ExecutorService` 都有效。任意的 `x` 很好理解，这和在 `unit` 以及 `map2`（或者其他由此衍生的组合子）中的应用的场景是一样的。那 `ExecutorService` 呢？它同等可能的实现会是怎样的呢？类 `java.util.concurrent.Executors`（API 链接：http://mng.bz/urQd）里罗列了各种不同的实现。

◇ **练习 7.8**

难：过一下 `Executors` 所有的静态方法，大致了解一下 `ExecutorService` 已有的实现的不同。然后，在继续之前，重温你在 `fork` 上的实现，设法找出一个反例，或说服自己法则对应你的实现是有效的。

为什么有关代码的法则和证据是重要的？

声明和证明 API 的性质似乎不是一种寻常的做法。至少在传统编程中不会这么搞。对于 FP 而言为什么这么重要呢？

在函数式编程中，我们可以想象到，将公共的功能分解到通用且可重用的组件是一件很容易的事情，这些组件之间还可以相互组合。副作用伤害组合性，但更普遍的是，任何隐藏或不可见（out-of-band）的假设或者行为都会妨碍我们将组件（无论是函数还是其他任何东西）视为一个黑盒，将它们进行组合不仅困难，甚至是不可能的。

在分流法则的例子中，如果法则无效，那么我们将发现很多通用的组合子，像 `parMap`，都不再合理啦（并且它们的用法可能是危险的，由于它们依赖更广泛的并行计算，最终导致死锁）。

赋予 API 代数的性质，在法则的应用下，它们变得更有意义且有助于推导，对用户而言更可用，同时也意味着我们可以把 API 的对象视为黑盒子。在第三部分我们将会看到，从不同的库之间分解出公共的模式，对我们的能力而言至关重要。

实际上会有一个相当微妙的问题出现在大多数 fork 的实现上。当使用固定大小的线程池作为 ExecutorService（见 Executors.newFixedThreadPool）时，是很容易出现死锁的。[14] 比如 ExecutorService 背后的线程池大小为 1，尝试运行下面的例子：

```
val a = lazyUnit(42 + 1)
val S = Executors.newFixedThreadPool(1)
println(Par.equal(S)(a, fork(a)))
```

大多数 fork 的实现都会在上面的代码中遭遇死锁。你能看出来为什么吗？让我们来回顾一下 fork 的实现：

```
def fork[A](a: => Par[A]): Par[A] =
  es => es.submit(new Callable[A] {
    def call = a(es).get ←——等待结果的Callable嵌套在了另外一个Callable中。
  })
```

请注意，我们在提交第一个 Callable 时，在它里面又向 ExecutorService 提出另一个 Callable，阻塞其结果返回（不要忘记 a(es) 将提交一个 Callable 到 ExecutorService，并得到一个 Future）。一旦线程池的大小为 1，这就成了问题。处在外面的 Callable 发起的提交阻塞了唯一的线程。就在这个线程里，直到完成之前，我们提交并阻塞等待另一个 Callable 的结果。与此同时却没有可用线程去运行这个 Callable。它们之间相互等待，于是死锁发生了。

◇ 练习 7.9

难：基于现在 fork 的实现，证明在任何固定大小的线程池都会出现死锁。

当你找到这样的反例时，你有两个选择，要么尝试修复实现，使得法则成立；要么完善法则，声明它有效的前提条件（你可以简单地规定线程池可以无限增长）。这是个不错的做法，它迫使你文档化以前隐含的不变量或假设。

我们可以修复这个 fork 使其能够在固定大小的线程池下工作吗？让我们来看看不同的实现：

```
def fork[A](fa: => Par[A]): Par[A] =
  es => fa(es)
```

这无疑可以避免死锁。唯一的问题是，我们实际上并没有派生一个单独的逻辑线程来对 fa 进行求值。所以，对于一些 ExecutorService es 而言，fork(hugeComputation)(es) 将在主线程中运行 hugeComputation，其实是想避免调用 fork。无论如何，这仍然是一个有用的组合子，因为它让我们做到延迟计算实例，直到它被真正地需要。让我们给它一个更合适的名字，delay：

```
def delay[A](fa: => Par[A]): Par[A] =
  es => fa(es)
```

但是，我们真的很想能够在固定大小的线程池上运行任意计算。为了做到这一点，我们需要为 Par 挑选另一个表示类型。

---

14 在下一章中，我们会写一个用于测试的组合子库，它可以自动帮助发现这样的问题。

7.4.4 用 Actor 实现一个完全无阻塞的 Par

在本节中，我们将基于固定大小的线程池实现一个完全无阻塞的 Par。对于函数式设计而言，本节的讨论不是必需的，如果你愿意可以跳过这一节。否则，请继续阅读。

现有表现类型的问题在于无法从 Future 获取值时还不阻塞当前线程。也就是说，Par 的表示类型必须是无阻塞的（non-blocking），即 fork 和 map2 的实现中不能有类似 Future.get 这样阻塞线程的方法调用。写这样一个正确的实现是很有挑战性的。幸运的是，我们有法则去测试我们的实现，而且仅需要生效一次就行。在那之后，库的用户可以享受一个可组合的且抽象的 API 去做对每一件事。

对于下面的代码，你并不需要了解其中每个部分到底发生了什么。我们只是想用真实的代码告诉你，满足法则的 Par 的表示类型是什么样的。

基本思路

怎样才能实现一个非阻塞的 Par 表示类型？这个想法很简单。用我们自己的 Future 版本，它可以注册一个 *Callback*，在结果准备好后被回调，替换之前的 java.util.concurrent.Future 获取值的方式（这会发生阻塞）。这是一个视角的转换：

```
sealed trait Future[A] {                    apply方法只能在fpinscala.parallelism包内被访问。
  private[parallelism] def apply(k: A => Unit): Unit
}                                                  Par看上去是一样的，只是用了我们自己的Future。
type Par[+A] = ExecutorService => Future[A]
```

我们的 Par 类型看起来一样，只不过现在使用新的 Future，与 java.util.concurrent 版本不同。与其从 Future 中 get 结果值，不如通过 apply 方法接收一个函数 k，它同样接收 A 类型的结果值，并对其执行一些操作。这种函数有时被称作 *continuation* 或 *callback*。

apple 方法被标记 private[parallelism]，为的是不暴露给库的使用者。标记为 private[parallelism] 确保了它只能被 fpinscala.parallelism 包内的代码访问。这是为了让我们的 API 保持 pure，同时也保证了法则的成立。

对 Pure API 使用局部副作用

Future 这样定义是相当有必要的。一个 A => Unit 这样的函数，只能被用于对给定的 A 执行一些副作用的操作，而且我们很清楚它不会有返回结果。我们仍然在使用像 Future 这样的类型进行函数式编程吗？是的，但我们正在利用使用副作用这样的常见技术来实现纯函数式 API 的细节。我们可以忽略掉它，因为涉及副作用部分在使用 Par 时是不可见的。不要忘了 Future.apply 是被保护的，不会被外部的代码调用到。

通过余下的非阻塞实现，你可能会说服自己，涉及副作用的部分对外部代码而言是不可见的。这种局部效应、可观测性，以及定义纯度和引用透明的微妙之处将在第 14 章详细讨论，但现在只需要有个概念就好。

基于新的 Par 表示类型，让我们看看如何实现 run 函数，为此我们要将返回值类型改为 A。既然已经从 Par[A] 到 A，就需要构造一个 continuation 并将它传递给 Future 的 apply 方法。

示例 7.6 实现 run

```
def run[A](es: ExecutorService)(p: Par[A]): A = {
    val ref = new AtomicReference[A]   ←— 可变但线程安全的引用，用于存储结果值。更多信息详见java.util.concurrent.atomic。
    val latch = new CountDownLatch(1)   ←— CountDownLatch会使得当前线程等待，直到countDown方法被调足期望的次数。在这里用来阻塞，以实现对异步计算结果的获取。
    p(es) { a => ref.set(a); latch.countDown }   ←— 在得到结果时，将其set给ref，并释放latch。
    latch.await   ←— 等待直到ref被set。
    ref.get   ←— 一旦latch被释放则获取结果。
}
```

应当指出的是，run 在等待 latch 的时候会阻塞调用线程，即阻塞是不可避免的。因为它需要返回类型 A 的值，所以它必须等待该值变得可用再返回。出于这个原因，我们希望 API 的用户避免调用 run，直到他们确定想等待这个结果。我们完全可以删掉 run，直接将 apply 方法暴露给用户，由他们来注册异步的回调。这肯定是一个有效的设计选择，但目前我们保持这个现状。

让我们来看一个实际创建 Par 的例子。最简单的肯定是 unit 了：

```
def unit[A](a: A): Par[A] =
    es => new Future[A] {
        def apply(cb: A => Unit): Unit =
            cb(a)   ←— 直接将a传递给continuation。注意这里是不需要ExecutorService的。
    }
```

由于 unit 已拥有 A 类型的值，它需要做的就是调用 continuation cb，传递这个值给它。如果 continuation 作为 run 的实现，它会释放 latch，使得结果立即可用。

fork 的实现会是怎样的呢？这里才是我们实际引入并行计算的地方：

```
def fork[A](a: => Par[A]): Par[A] =
    es => new Future[A] {
        def apply(cb: A => Unit): Unit =
            eval(es)(a(es)(cb))   ←— eval分流求值a并立即返回结果，cb将在另外线程中被异步调用。
    }

def eval(es: ExecutorService)(r: => Unit): Unit =   ←— 一个helper函数使用ExecutorService执行异步调用。
    es.submit(new Callable[Unit] { def call = r })
```

当 fork 返回 Future 接收到 cb 时，它会在分流后去对参数 a 进行求值。一旦参数被求值，则调用产出了 Future[A]，我们注册 cb 在 Future 有了结果 A 时被调用。

那 map2 又怎么实现呢？回想一下签名：

```
def map2[A,B,C](a: Par[A], b: Par[B])(f: (A,B) => C): Par[C]
```

这里要实现无阻塞，那是相当棘手。从概念上讲，我们希望 map2 并行运行这两个 Par 参数。当两个结果都抵达时，我们要调用 f，然后返回将生成的 C。但有几个竞争条件（race condition）需要注意，毕竟仅用 java.util.concurrent 里的低级操作实现正确的非阻塞是很困难的。

简要介绍 Actor

要实现 map2，我们将使用叫 actor 的无阻塞并发实现。Actor 基本上是并行处理的，且不中断线程。相反，它仅在接收消息的时候占用一个线程。重要的是，虽然多个线程可同时将消息发送给一个 actor，但它一次仅处理一个消息，其他的消息需排队供后续处理。当多线程必须访问那些棘手的代码时，这种做法将对并发实现非常有用，否则很容易出现竞争条件或死锁。

这最好用一个例子来说明。很多 actor 的实现都能满足我们的需求，其中包括 Scala 的标准库中的实现（见 scala.actors.Actor）。简单起见，我们将使用精简的 actor 实现，其源码在文件 Actor.scala 中：

```
scala> import fpinscala.parallelism._

scala> val S = Executors.newFixedThreadPool(4)
S: java.util.concurrent.ExecutorService = ...

scala> val echoer = Actor[String](S) {
     |   msg => println (s"Got message: '$msg'")
     | }
echoer: fpinscala.parallelism.Actor[String] = ...
```

Actor使用ExecutorService处理收到的消息，因此这里我们需要创建一个。

这儿Actor的实现很简单，就是打印收到的消息。请注意，S是一个用来处理消息的ExecutorService。

让我们来尝试这个 Actor：

```
scala> echoer ! "hello"
Got message: 'hello'

scala>

scala> echoer ! "goodbye"
Got message: 'goodbye'

scala> echoer ! "You're just repeating everything I say, aren't you?"
Got message: 'You're just repeating everything I say, aren't you?'
```

发送hello给Actor。

这里echoer并没有占用当前线程。

发送"goodbye"给Actor，Actor再次提交任务给ExecutorService处理消息。

这里完全没有必要去理解 Actor 的实现。正确且有效的实现是相当微妙的，你要是很好奇，见本节代码 Actor.scala 文件，其实现只有 100 行普通的 Scala 代码。[15]

通过 Actor 实现 MAP2

现在，我们可以一个 Actor 来收集这两个参数的结果以实现 map2 了。代码很简单，而且不用担心竞争条件，因为我们知道，Actor 一次只处理一条消息。

15 实现 Actor 最麻烦的在于多个线程可能同时发送消息给 Actor。实现需要确保消息在同一时间只有一个被处理，并且发送到 Actor 的消息将被最终处理，而不是无限期排队。即使这样，代码最终还是简短的。

示例 7.7 用 Actor 实现 map2

```
def map2[A,B,C](p: Par[A], p2: Par[B])(f: (A,B) => C): Par[C] =
    es => new Future[C] {
        def apply(cb: C => Unit): Unit = {
            var ar: Option[A] = None
            var br: Option[B] = None
            val combiner = Actor[Either[A,B]](es) {

                case Left(a) => br match {
                    case None => ar = Some(a)
                    case Some(b) => eval(es)(cb(f(a, b)))
                }

                case Right(b) => ar match {
                    case None => br = Some(b)
                    case Some(a) => eval(es)(cb(f(a, b)))
                }
            }

            p(es)(a => combiner ! Left(a))
            p2(es)(b => combiner ! Right(b))
        }
    }
```

两个可变的var是用来分别存储两个结果。

actor用来将等待的结果应用f，并最终传递给cb。

A的结果先到，则等B的结果。A的结果后到，调用f将结果C给回调cb。

同上，B的结果先到，则等A的结果。B的结果后到，调用f将结果C给回调cb。

用Left封装A，用Right封装B，并将二者发给actor。Either类型在此的用意是说明结果的来源。

基于这样的实现，我们再也不必担心用完线程，可以任意 `run` 各种复杂的 `Par` 了，尽管 Actor 只访问了一个 JVM 线程。

让我们在 REPL 中尝试一下：

```
scala> import java.util.concurrent.Executors

scala> val p = parMap(List.range(1, 100000))(math.sqrt(_))
p: ExecutorService => Future[List[Double]] = < function >

scala> val x = run(Executors.newFixedThreadPool(2))(p)
x: List[Double] = List(1.0, 1.4142135623730951, 1.7320508075688772,
2.0, 2.23606797749979, 2.449489742783178, 2.6457513110645907, 2.828
4271247461903, 3.0, 3.1622776601683795, 3.3166247903554, 3.46410...
```

这将调用 `fork` 10 万次，启动了很多 Actor 实现一次合并两个值。凭借 Actor 的非阻塞实现，我们不需要用 1 万个 JVM 线程来跑。

这样太赞了。我们分流的法则现在对固定大小的线程池也是有效的啦。

◇ **练习 7.10**

难：无阻塞表现类型目前并未处理错误。一旦计算抛出异常，`latch` 则永远不会计数，连异常也被简单地掩盖了。你能解决这个问题吗？

退一步说，这一节的目的并不是一定要弄清楚 fork 的最佳非阻塞实现，而是想表明法则的重要性。它让我们能够换个角度考虑库的设计。如果不是写下关于 API 的法则，我们不会那么早就发现之前实现的线程泄漏问题。

通常情况下，你有很多种方式选择 API 的法则。你可以思考你的概念模型，并从中推导出基本的法则。你也可以创造你觉得可能有用的或指导性的法则（如同我们创建分流法则一样），看看是否可能且合理地确保它们对你的模型有效。最后，你可以审视你的实现，并从中发现期望中的法则。[16]

7.5 完善组合子为更通用的形式

函数式设计是一个反复的过程。从写下 API，到至少有个原型实现，它都被不断地应用在越来越复杂和真实的场景中。有时候你会发现，这些场景中需要新增组合子。但立即实现之前，尝试将你需要的组合子完善到更通用的形式则更为明智。很有可能你所需要的只是通用的组合子的一个特殊用法。

关于本节的练习

本节中的练习和答案用的是最初简单的（阻塞）Par[A] 的表示类型。如果你想看看非阻塞的实现，可以看文件 Nonblocking.scala。

让我们来看一个这方面的例子。假设我们想要的函数是基于一个计算的结果来二选一地 fork 后面的计算：

```
def choice[A](cond: Par[Boolean])(t: Par[A], f: Par[A]): Par[A]
```

如果第一个计算结果为真，则选择处理 t，反之则是 f。实现肯定是等待 cond 的结果，再决定是要执行 t 还是 f，因此这里有个简单的基于阻塞的实现：[17]

```
def choice[A](cond: Par[Boolean])(t: Par[A], f: Par[A]): Par[A] =
  es =>
    if (run(es)(cond).get) t(es) ←——注意我们会在cond的结果上发生阻塞。
    else f(es)
```

也许你觉得这样已经不错，可是你再仔细想想这个组合子，它在干什么？它先运行 cond，待结果可用的时候运行 t 或 f。这似乎是合理的，但是我们却可以从其中的变化发现这个组合子的本质。这里有相当随意的布尔值，而事实上，我们只在两种可能的并行计算 t 和 f 上做选择。为什么只有两个？如果它能基于第一结果而对两个平行计算之间进行选择，那也能对 *N* 个计算进行选择：

```
def choiceN[A](n: Par[Int])(choices: List[Par[A]]): Par[A]
```

比起二选一，这种根据第一个结果 n 对更多计算进行选择的 ChoiceN 更为通用。

16 最后的产生法则的方法也许是最弱的，由于它太容易而不能很好地映射到你的实现上，甚至你的实现到处是 bug 和限制条件使这件事变得更难。

17 非阻塞的版本见 Nonblocking.scala。

◈ 练习 7.11

实现 choiceN。

到目前为止，我们完善最初的组合子，从 choice 到 choiceN，它变得更通用了，不仅能够做 choice 能做的，还能做 choice 不能做的。然后，还可以做得更通用。

◈ 练习 7.12

choiceN 的定义还是太草率了，过早地限定在 List 类型上。容器类型在这儿真的重要吗？比如，我们想一个 Map 实现替代列表的实现：[18]

```
def choiceMap[K,V](key: Par[K])(choices: Map[K,Par[V]]): Par[V]
```

只要你愿意，你可以停下来不再阅读后文，而是自己尝试实现这个新的更通用的组合子。

对 Map 的选择其实也过于具体，就像 List。仔细看一下 choiceMap 的实现，我们会发现其中没有多少用到 Map 的 API。其实，Map[A, Par[B]] 相当于函数 A =>Par[B]。到现在，不妨回顾一下 choice、choiceN，若是将 choice 的参数配对，其实就是函数 Boolean => Par[A]，而 choiceN 就是函数 Int => Par[A]！

让我们统一定一个更通用的签名吧：

```
def chooser[A,B](pa: Par[A])(choices: A => Par[B]): Par[B]
```

◈ 练习 7.13

实现 chooser，然后用它实现 choice 和 choiceN。

无论何时你用这样的方式泛化函数，一旦完成一定要用批判的眼光审视一下。虽然函数可能被用于某些特定的场景，但签名和实现却可以具有更普遍的意义。在这种情况下，chooser 相对于它的操作而言也许不再是最合适的命名啦，实际上它是相当通用的一个并行计算，一旦运行，则是先计算前置值，再由它决定下一步的计算。此外，并没有要求说，第二步的计算结果就一定要在第一步计算之前就得存在，又或者说一定要存储在某个容器里，如 Map 或 List，说不定整个结果都是根据第一个计算的结果算出的。这个函数，函数式库中常常有，一般被称为 bind 或 flatMap：

```
def flatMap[A,B](a: Par[A])(f: A => Par[B]): Par[B]
```

flatMap 真的就是最基本的函数了吗？我们还能更进一步泛化吗？不妨让我们再试试。flatMap 这个名字告诉我们，此操作可被分解为两个步骤：在 Par[A] 基础上映射（mapping）f: A => Par[B] 生成 Par[Par[B]]，然后再压扁（flattening）这个嵌套 Par[Par[B]] 到 Par[B]。这意味着我们需要一个更简单的组合子，就叫它 join 吧：

```
def join[A](a: Par[Par[A]]): Par[A]
```

18 Map[K, V]（API 链接：http://mng.bz/eZ4l）是 Scala 标准库中一个纯函数式的数据结构。它将 K 型的键和 V 型值一一关联起来，并允许我们根据键来查找值。

现在它就在眼前，我们便能仔细思考签名到底意味着什么，我们再一次仅根据类型就做到了，根据给定的函数签名，我们就能把它写出来。现在它就在眼前，我们便能仔细思考签名到底意味着什么。我们之所以叫它 join，是因为概念上它是一个并行计算，一旦运行，将执行内部计算，并等待其结束（有点像 Thread.join），最后返回结果。

◈ **练习 7.14**

实现 join，再用它实现 flatMap。

尽管我们在这里点到为止了，但你还可以在代数模型上进一步完善函数，尝试更复杂的例子，发现新的组合子，看看你能做到什么程度！下面留了一些问题供你思考：

- 你可以用 flatMap 和 unit 实现 map2 吗？这与原来的实现有什么不同？
- 你能在其他的基本代数函数上想到与 join 相关的法则吗？
- 是否存在代数无法表达的并行计算？你能想到任何即便是增加了新的基本代数函数也无法描述的计算吗？

认识代数的表现力和局限性

函数式编程越写越多，你会慢慢练就一种本领，它能够让你识别出什么样的函数是可以用代数方式描述的，以及代数描述又有什么局限。比方说，在前面的例子中，类似 choice 这样的函数，我们可能没法像 map、map2 和 unit 一样，从一开始就能做到纯函数式的表达，也不太可能看出 choice 其实就是 flatMap 的一个特殊用法。随着时间的推移，这种识别过程会越来越快，构建必要的组合子以进行代数描述的能力会越来越强。这些技能将对所有的 API 设计工作有很大帮助。

从实践角度考虑，尽可能缩小 API 的范围是非常有好处的。关于这一点，早在基于已有组合子实现 parMap 时，我们就能够看得出来。这种情况常常出现，基础的组合子封装了一些特别的逻辑，重用它们即可避免重复逻辑。

7.6 小结

现在我们已经用纯函数式的方式设计好了一个专注定义并行和异步计算的库。虽然这个领域很吸引人，但本章的主要目的是让你了解函数式设计的过程，以及你极有可能遇到的各式各样的问题和想法，并知道如何去处理它们。

本书从第 4 章到第 6 章都在讨论一个分离焦点（separation of concerns）的主题：具体来讲，就是将计算的描述与实现分离。在本章中亦是如此，在库的设计中，我们将并行计算的描述抽象为 Par，而实现则分离到了 run 当中。

在下一章中，我们将进入到一个完全不同的领域，并开启另一段曲折的旅程，更进一步地学习函数式设计。

8 基于性质的测试

在第 7 章中已经完成了用于表达并行计算的函数式库的设计。其中，我们引入 API 应该是代数形式的观点，具体而言，它表现为一批数据类型和基于这些类型的函数，以及更重要的那些表达了函数彼此关联的定律和性质。我们隐晦地表示了自动化验证这些定律的可能。

本章我们就来聊一聊这样一个简单而又强大的基于性质的测试库。一般而言，这样的库是把编程行为说明与测试用例的创建分离的。让程序员聚焦于阐述程序的行为，抽象测试用例的约束；而后框架会自动生成满足约束的测试用例，并运行测试来判定程序行为是否满足预期。

照理而言一个用于测试的库和一个并行计算的库应该有很多不同，但后面的内容会让我们发现它们有着惊人相似的组合件。类似情况在第三部分还会见到。

8.1 基于性质测试概览

下面是基于 ScalaCheck（http://mng.bz/n2j9）的例子，它是一个基于性质的测试库，它的一个性质看上去应该是这样的。

示例 8.1 ScalaCheck 特质

```
val intList = Gen.listOf(Gen.choose(0,100))   ←从0到100中生成一个整数列表。
val prop =                                    ←声明一个指定List.reverse方法的行为性质。
                                                检查当被reverse两次后应该恢复到最初值。
  forAll(intList)(ns => ns.reverse.reverse == ns) &&
                                              检查头一个元素与reverse后的最后一个元素是一样的。
  forAll(intList)(ns => ns.headOption == ns.reverse.lastOption)

val failingProp = forAll(intList)(ns => ns.reverse == ns)   ←一个明显会失败的性质。
```

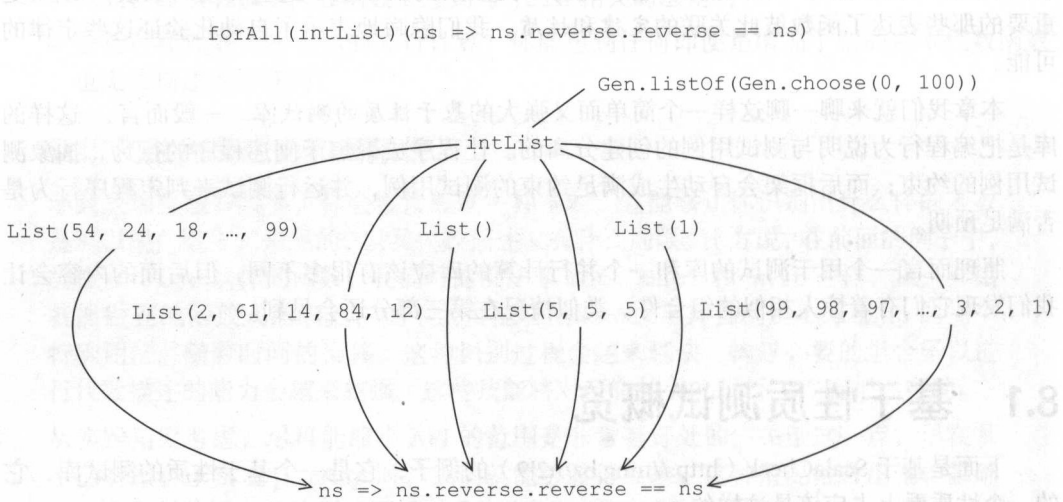
然后我们可以这样来验证性质：

```
scala> prop.check
+ OK, passed 100 tests.
```

```
scala> failingProp.check
! Falsified after 6 passed tests.
> ARG_0: List(0, 1)
```

这里，intList 不是 List[Int] 而是 Gen[List[Int]]，一个知道如何生成类型为 List[Int] 的测试数据的东西。我们可以从这个生成器中获得样例数据，它们是由 0 到 100 中随机填满的不同长度的列表。在基于性质的测试库中生成器有着丰富的 API，我们可以以不同的方式合并组合它们、重用它们，等等。

函数 forAll 就是通过组合 Gen[A] 和 A => Boolean 来创建性质的，进而验证所有生成器产生的值满足其中的论断。和生成器一样，性质也有着丰富的 API，比方说上面这个简单的例子中的 &&，它将 reverse 方法的两个性质合并起来，去验证生成的任何测试用例，直到找到不满足期望的情况。[1]



Gen对象生成一系列不同的值，并传递给一个条件表达式，最终找出不满足的情况。

当我们调用 prop.check 时，ScalaCheck 将会随机生成 List[Int] 去尝试找出我们论断错误的可能情况，最后的结果显示 ScalaCheck 生成了 100 个测试用例且全部通过验证。性质当然也会验证失败，就如 failingProp.check 结果所见，其中关于输入的信息对我们进一步测试或调试非常有用。

◈ 练习 8.1

习惯上面的测试方法，并实现 List[Int] => Int 这样一个求和函数的性质。不一定要写能够执行的 ScalaCheck 代码，能够表达意思就好。这里有一些点子供你参考：

1 这些测试的目的不是要覆盖所有程序的行为，而是为了提升代码的信心。和其他测试一样，我们可以让性质覆盖得更完整，但需要从投入产出比的角度考虑这样做是否值得。

- 翻转列表求和与原始列表求和的结果一致。
- 若列表的值都是一样的，其和应该是什么？
- 你还能想出其他的性质吗？

◈ 练习 8.2

一个求 `List[Int]` 的最大值的性质是什么？

基于性质的测试库通常还有其他一些特性。我们会在后面展开讨论，这里仅大致提一下：

- *测试用例最小化*——一旦测试失败，框架便尝试缩小范围直到明确导致失败的最小用例集合，以让调试更容易。比如，当列表长度是 10 时性质会失败，那框架则会试着缩小列表的长度，直到最小的情况被发现，最后告知我们。
- *穷尽测试用例*——某些情况下，`Gen[A]` 的空间（domain）[2] 足够小（比如不超过 100），我们会穷尽所有值来测试，而不是其中一部分样例数据。即当性质可以囊括所有空间的值时，我们则可在无任何疏漏的情况下证明我们的论断。

ScalaCheck 是基于性质测试库实现之一，它没什么不好的，只是我们会在本章从无到有地实现一个我们自己的测试库。和我们在第 7 章的考虑一样，更多是为了教学的目的，也是想表明没有任何库是不可替代的。使用像 ScalaCheck 这样已有的库本身没什么问题，而且它们也是不错的思路来源。即便你认为现有库的解决方案不错，也需要花一两个小时把玩一下它们的设计，写下一些类型签名，以便了解得更多，明白其中设计上的取舍。

8.2 选择数据类型和函数

本节又将是一个混乱迭代的过程，去发现库所适合的数据类型和函数。不同的是，我们将要设计一个基于性质的测试库。和之前一样，这又是一次站在别人肩膀上完成整个过程的机会。我们所实现的方法，以及库最终的样子没必要和你的完全一样。如果你还不太熟悉基于性质的测试，那样也许更好，这可以让你探索一个全新的领域，收获属于自己的发现。一旦你受到了启发，或有自己喜欢的点子，不要等到练习的时候，立马就可以动手去实现它。要是没了思路或搞不定了，完全可以再回到章节里来。

8.2.1 API 的初始代码片段

说了这么多，还是动手吧。什么样的数据类型是我们在测试库中该用的呢？我们该定义哪些 primitives，而它们的含义又是什么？函数应该满足什么样的定律？这一切还是

2 这里的“空间”（domain）与函数空间一样（http://mng.bz/ZP8q）——生成器描述了我们想要测试的函数的可能输入。注意，我们还会在一些口语化的场景中使用“空间”，指代一个主题或有趣的领域，比如函数式并行空间，或是错误处理空间。

从一个简单的例子入手吧，记下所需的数据类型和函数，看看我们能找到什么。之前那个ScalaCheck的例子挺有启发的：

```
val intList = Gen.listOf(Gen.choose(0,100))
val prop =
    forAll(intList)(ns => ns.reverse.reverse == ns) &&
    forAll(intList)(ns => ns.headOption == ns.reverse.lastOption)
```

尽管不清楚`Gen.choose`和`Gen.listOf`具体是怎么实现的，但我们仍能猜出它们返回的应该是一个参数化的类型（比方说`Gen`，生成器的简称）。因此，`Gen.choose(0,100)`极有可能返回`Gen[Int]`，而`Gen.listOf`返回的是函数`Gen[Int] => Gen[List[Int]]`。

然而`Gen`接收什么类型的参数其实不应该是`Gen.listOf`关心的（它可能需要另外的组合子来创建不同的列表，如`Int`、`Double`、`String`等），那我们就这样定义吧：

```
def listOf[A](a: Gen[A]): Gen[List[A]]
```

从签名上便能看出不少信息。注意，我们并没有指定列表应该生成的长度。为了方便实现，我们的生成器要么假定一个大小，要么被告知具体的大小。假定一个大小看起来不怎么灵活，毕竟任何假定都不太可能满足所有的情况。那么被告知具体的大小才是合适的做法。我们可想象API会是这样的：

```
def listOfN[A](n: Int, a: Gen[A]): Gen[List[A]]
```

这肯定是个非常有用的组合子，即便没有显性地指定大小它也同样强大。这意味着无论什么函数执行测试都可以自由地选择测试用例的大小，这样让前文提到的最小测试用例情况变成了可能。相反大小被程序员定死，则运行测试就丧失了灵活性。保持这样的关注点会让我们的设计走得更远。

例子剩余的部分会是怎样的呢？`forAll`函数看起来挺有趣的。可以看到它接收一个`Gen[List[Int]]`和一个相应的predicate `List[Int] => Boolean`。同样的`forAll`也不应该关注生成器和predicate的类型，只要它们是一样的就行。泛化之后可以是这样的：

```
def forAll[A](a: Gen[A])(f: A => Boolean): Prop
```

这里我们简单地引入了一个新类型`Prop`（`property`的简写，沿用ScalaCheck的命名）作为函数的返回值。我们尚不知道`Prop`的内容表现形式，或它支持哪些函数，但基于上述的例子我们发现它是有操作符`&&`的，那么它至少是这样的：

```
trait Prop { def &&(p: Prop): Prop }
```

8.2.2 性质的含义与API

目前我们已经有了一些API了，接下来就讨论一下这些类型和函数的含义。首先来看`Prop`，已经存在的函数有：

- `forAll`用来创建性质；
- `&&`用来组合性质；
- `check`用来运行性质。

在 ScalaCheck 中，check 方法存在输出控制台的副作用。为了方便这样定义函数没什么大问题，就是不利于组合。比方说，这样的 check 方法我们是没法实现 && 的：[3]

```
trait Prop {
  def check: Unit
  def &&(p: Prop): Prop = ???
}
```

由于 check 的副作用，使得实现 && 的唯一方式是二者都要运行。如果 check 会打印测试报告，则我们会得到两份报告，其中的成功和失败情况是彼此独立的。

这看起来不是一个正确的实现。check 的副作用问题是其次，更重要的是它丢掉的信息。

为了能够用像 && 的函数组合 Prop，我们需要让 check（或任何运行性质的函数）返回一个有意义的值。那这个值的类型是什么呢？呃，让我们想想要从检查性质中得到的那些信息。至少性质是成功还是失败是我们必须要知道的，先实现 && 再说。

◈ 练习 8.3

假定 Prop 的定义是下面这样的，请实现方法 &&。

```
trait Prop { def check: Boolean }
```

这样的定义里，Prop 有点像 Boolean，而且任何 Boolean 的函数（AND、OR、NOT、XOR 等）都能够被定义用于 Prop。但 Boolean 并不意味着 Prop 的全部，一旦性质失败，我们会想知道先前成功的测试有哪些，以及什么样的参数导致失败。又或者性质成功，知道运行多少测试是很有帮助的。不如尝试返回一个 Either 表示不是成功就是失败：

```
object Prop {
  type SuccessCount = Int ←——定义一个类型别名增加可读性。
  ...
}

trait Prop { def check: Either[???,SuccessCount] }
```

失败的场景下又该返回什么类型呢？关于生成的测试用例的类型我们一无所知。那加个类型参数让它变成 Prop[A]？ check 则返回 Either[A,Int]。但在此之前，让我们扪心自问一下我们关心性质失败的类型么？然而并非如此。我们只有在要对失败做进一步操作时才需要关心这个类型。最有可能的是为了能够检查测试的结果而打印到屏幕就结束了。总而言之，我们的目标是找到 bug，并明示出导致测试失败的用例，以便别人能修复它。按照常规，我们不应使用 String 作为所需处理数据的类型，但它却是最容易让人看懂的。由此我们可以得到 Prop 的定义如下：

```
object Prop {
  type FailedCase = String
  type SuccessCount = Int
}
```

3 回想第 7 章中在 Thread 和 Runnable 的使用上我们也遇到了类似的问题。

```
trait Prop {
    def check: Either[(FailedCase, SuccessCount), SuccessCount]
}
```

在失败场景下，check 返回 Left((s,n))，其中 s 是导致失败的值的字符串，而 n 是失败前成功的次数。

目前我们一直在关注 check 的返回值，那它的参数又会是怎样的呢？目前，check 没有任何参数。这就够了吗？我们不妨看看 Prop 的创建方式，再来思考一下它将会用到信息。比方说 forAll：

```
def forAll[A](a: Gen[A])(f: A => Boolean): Prop
```

在不知道更多关于 Gen 定义的情况下，我们很难有足够的信息去生成 A（这也是我们在实现 check 中需要的）。因此现在需要我们将注意力转移到 Gen 上，从而更好地理解它，以及了解它的依赖关系。

8.2.3 生成器的意义和 API

先前我们定义了 Gen[A] 负责生成 A 类型的值，而这些生成的方法是什么呢？这可以是随机的生成。回想一下第 6 章的示例，我们实现了一个纯函数式数字生成器 RNG，并使其能够满足组合计算。这里我们可以将 Gen 设计成一个包装类型，其中封装了随机数生成器的状态机：[4]

```
case class Gen[A](sample: State[RNG,A])
```

◈ **练习 8.4**

用上面 Gen 的定义实现 Gen.choose 方法，生成指定范围内整数（不含右边界值）。可随意使用你已经实现的函数。

```
def choose(start: Int, stopExclusive: Int): Gen[Int]
```

◈ **练习 8.5**

基于这样的定义，请尝试实现 unit、boolean 和 listOfN。

```
def unit[A](a: => A): Gen[A]                    ←——总是生成值A。
def boolean: Gen[Boolean]
def listOfN[A](n: Int, g: Gen[A]): Gen[List[A]]  ←——基于生成器g生成长度为n的列表。
```

正如第 7 章中讨论的，我们更愿意发现哪些操作是原生的，而哪些操作是衍生的，以找到一个小且富于表现力的原生操作集合。探究这些原生操作是否具有表现力的一个好办法是，选一个具体的例子，看看它们是否能够按照你的意愿完成函数式的组合。在此过程中，发现模式，并尝试将它们提炼成组合子，最终精炼为那些原生操作。不妨这儿先停下阅读，把玩一下我们已实现的这些原生操作和组合子。若你想要一些具体的例子来启发自己，这里倒是有一些：

4　回想一下当时的定义：case class State[S,A](run: S => (A,S))。

- 既然我们能够从某个范围中生成一个 Int，那我们是否需要一个原生操作去基于一个范围生成 (Int, Int) 呢?
- 我们能够做到将 Gen[A] 转换成 Gen[Option[A]] 吗? 然后又能将 Gen[Option[A]] 还原成 Gen[A] ?
- 使用现有的原生操作我们能生成字符串吗?

把玩的重要性

你完全没有必要等待一个具体的例子去验证你的设计。事实上，一旦你依赖了一个特定的例子，即便它非常有用且很重要，而设计出来的 API 通常会遗漏一些方面，让 API 过于专用了。这种为特定例子的设计是我们不希望看到的。从本质上解决这个问题的最好办法就是把玩它们。不要尝试解决重要的问题或实现有用的功能，至少不是现在。只是试验各种定义、原生方法和操作，让问题自己冒出来，探究任何令你好奇的东西。（“这两个函数看起来很像，我想知道它们中是否有一个更抽象的操作”，或者“这个数据类型的多态性是否合理呢?”，再或者“要是把这个返回值定义由单值变为列表会意味着什么呢?”）这样做无对错之分，毕竟那么多设计选择会让你不顾一切投入到迷人的问题中。从哪儿开始并不重要——如果你一直玩，问题域会指引你做出必要的全部设计选择。

8.2.4 生成值决定生成器

假设我们想要 Gen[(String, String)] 生成的一对字符串中，第二个字符串中的字符必须是来源于第一个字符串的。或者我们有一个 Gen[Int] 能从 0 到 11 中选择一个整数，然后要实现一个 Gen[List[Double]] 去生成任何长度的列表。这两个场景中都有一个共性，我们先生成一个值，再根据这个值生成下一个值。这种情况下我们需要 flatMap，它可以让一个生成器依赖另一个生成器。

◈ **练习 8.6**

实现 flatMap，然后用它实现一个更动态的 listOfN。Gen 类中含有 flatMap 和 listOfN。

```
def flatMap[B](f: A => Gen[B]): Gen[B]
def listOfN(size: Gen[Int]): Gen[List[A]]
```

◈ **练习 8.7**

实现 union，将两个同类型的生成器组合成一个。

```
def union[A](g1: Gen[A], g2: Gen[A]): Gen[A]
```

◇ 练习 8.8

实现 weighted，类似 union 只是根据权重从不同的生成器中取得不同量的值。

```
def weighted[A](g1: (Gen[A],Double), g2: (Gen[A],Double)): Gen[A]
```

8.2.5 精炼 Prop 的数据类型

到目前为止我们知道不少生成器的定义信息，是该回到 Prop 的定义上来。看看 Gen 的定义其中已经展现 Prop 必要的信息，这使得 Prop 看起来是这样的：

```
trait Prop {
    def check: Either[(FailedCase, SuccessCount), SuccessCount]
}
```

Prop 就是一个不太严格的 Either，只是缺少了一些信息。尽管 SuccessCount 有了测试成功的个数，却没有指定多少个成功的测试才算通过，硬编码肯定是可以的，但把它变成参数会更好：

```
type TestCases = Int
type Result = Either[(FailedCase, SuccessCount), SuccessCount]
case class Prop(run: TestCases => Result)
```

另外，无论成功与否我们都记录下了成功的次数。可当性质测试通过时，测试成功的个数肯定是等于指定的参数的。这对测试的调用者而言毫无意义，因此对于 Eight 右边的情况我们现在完全不关心，还不如用 Option：

```
type Result = Option[(FailedCase, SuccessCount)]
case class Prop(run: TestCases => Result)
```

这看起来有点怪怪的，None 表示测试通过而 Some 表示测试失败。在此之前，我们一直是用 None 来表示失败的，仅在此用它来表示成功的情况。对 Option 而言这种用法无可厚非，只是容易让人混淆。为此我们创造一个新的数据类型，基本等同于 Option[(FailedCase,SuccessCount)]，这样一来就很明了了。

示例 8.2 创建一个 Result 数据类型

```
sealed trait Result {
    def isFalsified: Boolean
}
case object Passed extends Result {          ←——表明所有测试是否通过。
    def isFalsified = false
}
case class Falsified(failure: FailedCase,    ←——表明其中一个测试场景失败了。
                     successes: SuccessCount) extends Result {
    def isFalsified = true
}
```

这些足以表现 Prop 吗？我们再来看看 forAll，这样能否实现 forAll？或者为什么不能实现？

```
def forAll[A](a: Gen[A])(f: A => Boolean): Prop
```

我们可以看出 forAll 没有足够的信息以返回一个 Prop。除了要尝试多个测试用例外，Prop.run 需要其他所有用以生成测试用例的信息。比如需要基于 Gen 随机生成测试用例，那么就需要一个 RNG。我们先将这种依赖放到 Prop 上：

```
case class Prop(run: (TestCases,RNG) => Result)
```

如果还有其他可能的依赖，除了测试用例的个数和随机值的来源，我们可以把它们作为 Prop.run 的参数。

现在我们有足够的信息实现 forAll 了。下面就是一个简单的实现。

示例 8.3 实现 forAll

```
def forAll[A](as: Gen[A])(f: A => Boolean): Prop = Prop {
    (n,rng) => randomStream(as)(rng).zip(Stream.from(0)).take(n).map {
      case (a, i) => try {
        if (f(a)) Passed else Falsified(a.toString, i)
      } catch { case e: Exception => Falsified(buildMsg(a, e), i) }
    }.find(_.isFalsified).getOrElse(Passed)
}

def randomStream[A](g: Gen[A])(rng: RNG): Stream[A] =
    Stream.unfold(rng)(rng => Some(g.sample.run(rng)))

def buildMsg[A](s: A, e: Exception): String =
    s"test case: $s\n" +
    s"generated an exception: ${e.getMessage}\n" +
    s"stack trace:\n ${e.getStackTrace.mkString("\n")}"
```

在(a, i)中a是随机值，i是stream的数字下标。

当测试失败的时候，记录失败的用例和成功的个数。

若测试产生了异常则记录在结果中。

通过简单地重复调用生成器实现一个无限的A stream。

String interpolation语法。一个以s"开头的字符串其中对Scala值v的引用，如$v或${v}，Scala编译器都会自动将其转换成v.toString。

请注意我们捕获了异常并报告为测试失败，而不是任由异常乱抛（这样会丢失掉导致失败的参数信息）。

◈ **练习 8.9**

基于现在 Prop 的定义，实现能够组合 Prop 值的 && 和 ||。注意失败的场景中我们无法知道是由左边的性质还是右边性质导致的。能否设计一种方案处理这种情况，或是通过允许给 Prop 值打个标签以便失败的时候显示出来？

```
def &&(p: Prop): Prop
def ||(p: Prop): Prop
```

8.3 最小化测试用例

前文中我们提到了最小化测试用例的想法。理想中我们的框架能够找到最小测试用例集，或是最小的导致失败的测试用例集，以便更好地暴露问题，帮助调试。让我们试试微调一下我们的定义能否达到这个目的。通常我们有两种办法：

- 收缩——在发现失败的测试用例后，我们可以运行一个独立过程，继续缩小测试用例的大小直到没有失败。我们称这样的做法叫作收缩，它通常需要额外的编码来实现最小化的过程。
- 定长生成——与其事后收缩，不如简单地逐渐增加测试用例的长度和复杂度。也就说我们由小到大地直到发现失败。这种想法可以各种方式扩展使得测试在合理的长度范围内做大幅的跳跃，同时确定最小失败测试用例的集合。

顺带说一下，ScalaCheck 就是采用第一种收缩的方式实现的。用这种方式没什么不对的地方（ScalaCheck 所基于 Haskell 的实现 QuickCheck〔http://mng.bz/E24n〕也用了这种方式），只不过我们会用定长生成的方式来实现。这样实现起来更简单，某种程度上更模块化，因为生成器只需要关心如何生成给定大小的测试用例，不需要操心查找测试用例的空间的 schedule，且运行的函数可以自由地选择 schedule。你会发现实现相当精简。

在不改变 Gen 的数据类型情况下，有了这些有用的已经实现的组合子，我们只需要将定长生成作为独立的一层引入我们的库。定长生成器的定义就是一个根据长度返回生成器的函数：

```
case class SGen[+A](forSize: Int => Gen[A])
```

◈ **练习 8.10**

实现一个帮助函数，能够将 Gen 转换成 SGen，并将此方法加到 Gen 上。

```
def unsized: SGen[A]
```

◇ **练习 8.11**

无任何意外，SGen 至少支持 Gen 的所有方法，且实现也是非常机械的。定义一些方便的函数使得 SGen 能够基于 Gen 实现。[5]

◈ **练习 8.12**

实现无须 size 参数的 listOf 组合子，它会返回一个 SGen。实现应该生成所需长度的列表。

```
def listOf[A](g: Gen[A]): SGen[List[A]]
```

让我们来看看 SGen 是如何影响 Prop 定义以及 Prop.forAll 的。SGen 版本的 forAll 是这样的：

```
def forAll[A](g: SGen[A])(f: A => Boolean): Prop
```

你能看出来为什么不可能实现这个函数吗？ SGen 是需要被告知长度的，但 Prop 不接收任何长度的信息。这和我们在处理随机性来源和测试用例数量非常相似，只需要简单将它们添加为 Prop 的依赖即可。因此让 Prop 接收一个最大长度，以便它能负责调用底

5 在第三部分，我们将讨论如何提炼这样的重复代码。

层的生成器。那么 Prop 将会逐渐生成测试用例直到满足给定的最大长度。同时也能支持查找出最小测试用例集。请看下面的实现。[6]

示例 8.4 逐渐生成测试用例直到给定最大值

```
type MaxSize = Int
case class Prop(run: (MaxSize,TestCases,RNG) => Result)

def forAll[A](g: SGen[A])(f: A => Boolean): Prop =
    forAll(g(_))(f)

def forAll[A](g: Int => Gen[A])(f: A => Boolean): Prop = Prop {
    (max,n,rng) =>
        val casesPerSize = (n + (max - 1)) / max ←——根据size生成对应的随机用例。
        val props: Stream[Prop] =
            Stream.from(0).take((n min max) + 1).map(i => forAll(g(i))(f)) ←——┐
        val prop: Prop =                                                    每个长度创建一个性质，但不超过n个。
            props.map(p => Prop { (max, _, rng) =>
              p.run(max, casesPerSize, rng)
            }).toList.reduce(_ && _) ←——组合成一个性质。
        prop.run(max,n,rng)
}
```

8.4 使用库并改进其易用性

基本上能想到的 API 我们都已囊括了，与其继续修修补补，不如尝试使用它们构造测试，看看还什么不足，主要是表达力或可用性上的。尽管可用性是一种很主观的东西，但我们希望它表现为简便的语法，加一些恰当的帮助函数以满足大部分的使用场景。我们并非要求库有多么的富于表现，但至少让它用起来很愉悦。

8.4.1 一些简单的例子

回顾一下在章节之初我们提到的一个例子——指明 List 的 max 方法的行为（API 文档链接：http://mng.bz/Pz86）。列表的最大值应该大于等于列表中任何一个其他的值。

```
val smallInt = Gen.choose(-10,10)
val maxProp = forAll(listOf(smallInt)) { ns =>
    val max = ns.max
    !ns.exists(_ > max) ←——ns中不会有值比max大。
}
```

到这里你可能已经发现直接调用 Prop.run 会比较麻烦。要是我们引入一个帮助函数在运行 Prop 之后随即将结果以合适的格式输出到控制台会好很多。这个方法可以放到 Prop 的伴生对象中。

---

6 最简单的实现方式就是从 0 开始，每次增加 1 的生成测试用例。更精明的做法是以类似二分查找方式去圈定失败测试用例的大小——如 0、1、2、4、8、16……逐渐缩小失败时间的搜索空间。

示例 8.5　运行 Prop 的帮助函数

```
def run(p: Prop,
        maxSize: Int = 100,  ←——参数的默认值为100。
        testCases: Int = 100,
        rng: RNG = RNG.Simple(System.currentTimeMillis)): Unit =
    p.run(maxSize, testCases, rng) match {
        case Falsified(msg, n) =>
            println(s"! Falsified after $n passed tests:\n $msg")
        case Passed =>
            println(s"+ OK, passed $testCases tests.")
    }
```

充分利用参数默认值可以让方法更易用。比如设定一个默认的测试个数，既有良好的测试覆盖，又不会因为太多而导致执行时间很长。一旦我们尝试运行 `run(maxProp)`，其结果会是失败的！基于性质的测试就是这样，可以暴露我们隐藏在代码中的某些假设，强迫我们去将它们显性地声明出来。标准库的 `max` 实现在给空列表的时候是会报错的。为此修正我们的性质时需要将这类情况考虑进去。

◈ **练习 8.13**

定义 `listOf1` 生成非空的列表，并基于它更正上面 `max` 的测试规格说明。

我们还可以尝试更多的例子。

◈ **练习 8.14**

编写一个性质验证 `List.sorted` 的行为（API 文档链接：http://mng.bz/Pz86），可以对给定的 `List[Int]` 进行排序。[7] 例如，`List(2,1,3).sorted` 的结果是 `List(1,2,3)`。

8.4.2 为并行计算编写测试套件

在第 7 章中我们发现了一些关于并行计算的法则。这些法则可否用我们的库来表达呢？我们发现的第一个法则就是一个特殊的测试用例：

```
map(unit(1))(_ + 1) == unit(2)
```

表达是肯定可以的，只是看起来有点丑。[8]

```
val ES: ExecutorService = Executors.newCachedThreadPool
val p1 = Prop.forAll(Gen.unit(Par.unit(1)))(i =>
    Par.map(i)(_ + 1)(ES).get == Par.unit(2)(ES).get)
```

7　`sorted` 使用 implicit `Ordering` 来控制排序的策略。

8　这儿的 `Par[A]` 可以看作是 `ExecutorService => Future[A]` 的别名。

正如所言，上面的代码是啰唆且混乱的，更重要的是核心的含义被不重要的细节所掩盖了。注意到没有，问题其实并不在 API 的表现力不够上，而是缺少恰当的帮助函数和灵活的语法。

证明性质

为此我们必须改进。首先 forAll 对于这个测试用例而言通用性太强，毕竟除了一个硬编码的例子外没有其他情况的输入了。硬编码的例子的编写就应该和传统的单元测试库一样简单方便。让我们引入一个组合子（属于 Prop 的伴生对象）来解决这个问题：

```
def check(p: => Boolean): Prop
```

如何实现它？其中一个可行的办法是使用 forAll：

```
def check(p: => Boolean): Prop = {  ←——这儿并非严格的实现。
  lazy val result = p  ←——将result作为中间变量避免重复计算。
  forAll(unit(()))(_ => result)
}
```

但这还是不够好，我们一边提供了一个 unit 生成器始终生成同一个值，另一边却又忽略这个值直接返回 Boolean 的结果。

尽管我们在保存了 result 以便它只会被计算一次，但测试用例会被生成多次，进而导致测试 Boolean 也会有多次。

比方说，我们执行 run(check(true))，那么实际测试会有 100 次，并且显示输出“OK, passed 100 tests.”。每次检查的结果都是 true，执行 100 次完全是白费的。我们需要的是一个新的原生操作。

Prop 的定义就是一个函数类型 (MaxSize, TestCases, RNG) => Result，其中 Result 要么是 Passed 要么是 Falsified。那 check 的简单实现无非是忽略掉那些参数：

```
def check(p: => Boolean): Prop = Prop { (_, _, _) =>
  if (p) Passed else Falsified("()", 0)
}
```

这显然要比 forAll 的实现好很多，只是 run(check(true)) 依旧会打印出“passed 100 tests”，哪怕测试只有一次。这并非真正的 true，只是一个性质通过了，则余下的测试都将是通过的，这种情况就叫被证明。这意味着我们需要一个新的 Result 种类：

```
case object Proved extends Result
```

然后我们用 Proved 替换 Passed 作为 check 的返回，并修改测试运行的代码支持这种情况：

示例 8.6 使用 run 返回 Proved object

```
def run(p: Prop,
        maxSize: Int = 100,
        testCases: Int = 100,
        rng: RNG = RNG.Simple(System.currentTimeMillis)): Unit =
```

```
    p.run(maxSize, testCases, rng) match {
        case Falsified((msg, n)) =>
            println(s"! Falsified after $n passed tests:\n $msg")
        case Passed =>
            println(s"+ OK, passed $testCases tests.")
        case Proved =>
            println(s"+ OK, proved property.")
    }
```

此外我们还需要修改 Prop 的 && 的实现。当然了，某些组合件并不区分 Passed 和 Proved，这让改动也很简单。

◇ **练习 8.15**

难：由于仅需要判断求 Boolean 值的参数，因而 check 很容易被证明。然而 forAll 也是存在可以证明的情况的。比如，在领域性质是 Boolean 时，它只有两种情况，如果对于 forAll(p) 而言 p(true) 和 p(false) 都测试通过，那它就是被证明了的。因此一些场景有限的领域（如 Boolean 和 Byte）是可以被穷举验证的。即便是无限情况的场景下，采用定长生成器也是可以通过穷举到最大长度来核实的。自动测试已经很强大了，要是我们能够做到自动证明代码的正确性就更了不起了。调整我们的库使之综合穷举验证和定长生成器两种算法，这将不光是一个练习，更是一个应用范围更广更开放的设计项目。

测试 PAR

回到先前证明 Par.map(Par.unit(1))(_ + 1) 等价于 Par.unit(2) 的问题上来，现在我们可以用 Prop.check 来实现啦：

```
val p2 = Prop.check {
    val p = Par.map(Par.unit(1))(_ + 1)
    val p2 = Par.unit(2)
    p(ES).get == p2(ES).get
}
```

这样就简洁不少吧。可对于 p(ES).get 和 p2(ES).get 这样的噪音我们能做什么改进吗？不然总是感觉不那么满意。比如说，仅仅只是判断两个 Par 的值是否相等，还要迫使代码关心 Par 内部实现的细节。一个优化的办法是将比较过程提升为对 map2 的调用，这就意味着直到运行最后的 Par 才能得出结果：

```
def equal[A](p: Par[A], p2: Par[A]): Par[Boolean] =
    Par.map2(p,p2)(_ == _)

val p3 = check {
    equal(
        Par.map(Par.unit(1))(_ + 1),
        Par.unit(2)
    )(ES).get
}
```

这要比单独运行两边要稍好一点。既然都做到这一步，我们为什么不将 Par 的运行放到一个独立的函数 forAllPar 中呢？它还是一个提供不同平行策略的好地方，且不会弄乱我们对性质的描述说明：

```
val S = weighted(  ←——分配75%的权重给定长线程池，余下25%给变长线程池。
    choose(1,4).map(Executors.newFixedThreadPool) -> .75,
    unit(Executors.newCachedThreadPool) -> .25)  ←——a -> b是(a,b)的语法糖。

def forAllPar[A](g: Gen[A])(f: A => Par[Boolean]): Prop =
    forAll(S.map2(g)((_,_))) { case (s,a) => f(a)(s).get }
```

S.map2(g)((_,_)) 对于组合两个生成器而言，看起来还是太累赘。让我们引入一个新的组合子让它变得简单点：[9]

```
def **[B](g: Gen[B]): Gen[(A,B)] =
    (this map2 g)((_,_))
```

这样好了很多：

```
def forAllPar[A](g: Gen[A])(f: A => Par[Boolean]): Prop =
    forAll(S ** g) { case (s,a) => f(a)(s).get }
```

甚至我们还可以引入一个 ** 抽取器（http://mng.bz/4pUc），它可以让代码变成这样：

```
def forAllPar[A](g: Gen[A])(f: A => Par[Boolean]): Prop =
    forAll(S ** g) { case s ** a => f(a)(s).get }
```

这种语法尤其适合于模式匹配中抽取元组的一对值，以避免我们使用嵌套括号的方式实现。使得 ** 变成一个模式的办法就是定义一个 ** 的 object，其中含有一个 unapply 的函数：

```
object ** {
    def unapply[A,B](p: (A,B)) = Some(p)
}
```

更多技术细节还请参看自定义抽取器的相关文档。

S 是一个 Gen[ExecutorService]，它可以不同比重来提供定长的线程池和不定长的线程池。这让目前的代码看起来清爽了不少：[10]

```
val p2 = checkPar {
    equal (
        Par.map(Par.unit(1))(_ + 1),
        Par.unit(2)
    )
}
```

它们看上去只是很小的改动，但就是这种分解和清理能大大改善库的可用性，还有那些帮助函数使得性质代码更易读，写起来也更容易让人心情愉悦。相信你现在也想实现一个 forAllPar 版本的定长生成器吧。

---

9 使用 ** 是非常合适的，因为函数被定义成输出两个生成器，这一点我们在第 3 章讨论过。

10 我们不能使用 Java/Scala 的 equals 方法，或者 Scala 中的 ==（其实就是 equals 方法），因为它是直接返回 Boolean，而我们需要的是 Par[Boolean]。使用中辍语法实现 equal 可能更合适，可以在第 7 章的练习答案中找到参考范例。

再来看看第 7 章其他的性质吧。其中一个会是这样的：

```
map(unit(x))(f) == unit(f(x))
```

在基于法则用 id 函数[11] 替换原有计算后得到了简化的版本：

```
map(y)(x => x) == y
```

这样的性质可以被测试描述（express）吗？可能做不到。这个等式暗指所有值和类型，而我们只能选择具体的值：

```
val pint = Gen.choose(0,10) map (Par.unit(_))
val p4 = forAllPar(pint)(n => equal(Par.map(n)(y => y), n))
```

尽管能赋予 y 更多值的选择，但这里的可能就已经足够了。map 的实现本身并不关心平行计算的值是什么，所以用 Double、String 或其他什么类型写类似的测试意义不大。真正影响 map 的是平行计算的结构。要想保证性质的有效性，我们就需要提供更丰富的结构生成器。只是这里我们仅提供一层嵌套的 Par 表达式。

◈ **练习 8.16**

难：编写一个 Par[Int] 的富生成器，它能生成比上面嵌套更深的平行计算。

◈ **练习 8.17**

尝试表达第 7 章关于 fork 的性质，fork(x) == x。

8.5 测试高阶函数及展望未来

到现在，我们的库看上去应该不错了，只是还有一个地方尚属空白：我们还没有一种好办法去测试高阶函数。尽管我们有很多方法生成*数据*，却没有办法生成*函数*。

举个例子，List 和 Stream 都定义了函数 takeWhile。调用它可以返回所有元素直到条件不被满足。比如 List(1,2,3).takeWhile(_ < 3) 的结果是 List(1,2)。一个简单的性质就是，对任何列表 s: List，任何函数 f: A => Boolean 表达式 s.takeWhile(f).forall(f) 的求值都为 true。也就是说结果中的每个元素都应该满足函数限定的条件。[12]

◈ **练习 8.18**

尝试提供其他的性质去验证 takeWhile。比方说思考如何用性质表达 takeWhile 和 dropWhile 之间的关系？

---

11 即 x => x。——译者注

12 在 Scala 标准库中 forall 是 List 和 Stream 中的一个方法，且签名是 def forall[A](f: A => Boolean): Boolean。

当测试高阶函数时，我们能做的是用具体的参数来验证。比方说，这个为 takeWhile 设计的具体性质：

```
val isEven = (i: Int) => i%2 == 0
val takeWhileProp =
    Prop.forAll(Gen.listOf(int))(ns => ns.takeWhile(isEven).forall(isEven))
```

这样虽然可行，但有没有一种方法让测试框架去为 takeWhile 生成函数呢？[13] 看看我们能做什么。好比说，我们有一个 Gen[Int]，希望得到一个 Gen[String => Int]，有什么好办法？我们可以通过简单地忽略字符串的参数，然后用 Gen[Int] 返回 Int 值：

```
def genStringIntFn(g: Gen[Int]): Gen[String => Int] = g map (i => (s => i))
```

这样做显然不太高效。我们还可以简化，生成一个忽略输入且返回常量的函数。在 takeWhile 的例子中，我们需要的是一个返回 Boolean 值的函数，那么提供一个始终返回 true 或 false 的函数不就可以了，毕竟 takeWhile 并不关心这个函数内部的行为。

◇ **练习 8.19**

难：若要让函数根据参数的值来确定返回的 Int 值，你有什么好办法呢？这是开放式的且很有挑战的设计练习，看看你能否发现其中的问题，并找到解决方案纳入到我们开发的库中。

◇ **练习 8.20**

我们非常鼓励你去探索和尝试使用我们开发的库！看看到底你可以拿它测试其他什么东西，说不定你还能发现新的点子，或是其他可进一步扩展或更易用的方法。这里提供一些切入点做参考：

- 写一些性质去描述 List 和 Stream 的其他行为，如 take、drop、filter，以及 unfold。
- 为第 3 章中 Tree 数据类型编写一个定长生成器，并使用它来验证我们定义的 fold 函数，然后再进一步思考如何改进 API 令它们更易用一些？
- 编写性质去描述序列函数的行为，如 Option 和 Either。

8.6 生成器法则

有没有发现一个有趣的现象，我们实现 Gen 类型的方法和之前定义的 Par、List、String 以及 Option 都很相似啊。比方说，在 Par 上我们定义了：

```
def map[A,B](a: Par[A])(f: A => B): Par[B]
```

而本章我们为 Gen 也定义了 map：

```
def map[B](f: A => B): Gen[B]
```

13 回忆一下在第 7 章中我们曾介绍过的自由定理，其中参数态（parametricity）使得函数行为与具体类型无关。这对于很多需要生成函数进行测试的情况仍然很适用。

同样，这些函数在 Option、List、Stream 和 State 上也有定义。之所以会这样，是否意味着它们之间满足一个同样的法则呢？就像第 7 章中应用在 Par 上的法则：

```
map(x)(id) == x
```

那是不是意味着 Gen.map 也满足这个法则呢？还有 Stream、List、Option、State？是的，它们都满足。不信可以试试看。这表明，它们不仅在签名上相似，在各自的问题域里也有着类似的含义。可见其内在的联系是多么紧密啊！我们正在揭示这些领域之间的基础模式。在第三部分中，我们将了解这些模式的名称，发现它们中的法则，进而理解它们的含义。

8.7 小结

本章我们进行了函数式库设计的另一种扩展练习，受到来自基于性质测试领域的启发。

我们一再强调学习基于性质的测试不是目的，而是突出函数式设计本身。首先，迂回在抽象代数与具体实现之间，可以达到一种平衡，既避免库的设计依赖特别场景，又避免抽象程度脱离了实际的目标无法落地。

其次，这个问题域让我们探索一些此前见过几次的组合子，像 map、flatMap 等。它们不光签名相似，连实现也都雷同。那些软件世界里看似不同的问题，它们的函数式解决方案却为数不多。很多库都是基础结构的简单组合，然后不断出现在不同的问题域里。这使得我们有机会进行代码重用，在本书的第三部分会专门深入这块，届时我们将认识这些结构，并学到如何发现更多通用的抽象。

接下来是第二部分的最后一个章节，我们将进入另一个领域——解析，那里有独特的挑战等着我们。尽管我们会在下一章采用一点不同的方法，但熟悉的模式依旧会重现的。

9 语法分析器组合子

本章我们将设计一个能够创建语法分析器的组合子库，并使用JSON的语法分析（http://mng.bz/DpNA）作为驱动场景。同第7、8章一样，本章重点是函数式设计而非语法分析。

什么是语法分析器？

语法分析器是一种特别程序，接收非结构化数据（比如文本，任何种类的符号、数字或字符的流）作为参数，输出结构化的数据作为结果。比方说，我们编写一个语法分析器将逗号分隔的文件转换成一个二维列表，每行是一条记录，每列表示逗号分隔的字段。再比如，编写一个语法分析器将XML或JSON文档变成一个树形数据结构。

对于本章中我们即将构建的语法分析器组合子库而言，语法分析器不是说要多复杂，不是非要分析全部的文档。相反，只要它能从输入中识别一个字符，我们便可通过组合子来组合简单的语法分析器成为复杂的语法分析器。

本章将介绍一种设计方法，我们叫它*代数设计*（algebraic design）。这是一个自然演化的过程，此前章节里我们在不同程度已经这样做了，先是设计接口和相关法则，再基于此选择其数据类型。

在本章的几个关键点上，我们将给予更多的开放式练习，旨在模拟从头开始编写自己的库时可能会遇到的场景。花些时间在练习上探索其他可能的方法，或许会让你学到更多本章之外的东西。在你设计自己的库期间，我们不会手把手地去教你实现，而是你自己做出类型和组合子的选择，这也是本书第二部分希望你达成的目标。照例，当你做练习遇到障碍，或是需要更多启发时，你可以继续阅读后文，也可以参考答案，还可以尝试与其他人一起做练习，在线上分享彼此的读书笔记。

语法分析器组合子与语法分析器生成器

你可能听说过像Yacc（http://mng.bz/w3zZ）这个语法分析器生成器库，或是其他语言的实现（比如用Java实现ANTLR〔http://mng.bz/aj8K〕）。这些库会根据语法规格说明生成语法分析器的实现代码。它们通常都运行得很好，而且执行效率

还不错，只是那一坨代码会让调试变得很困难。同样复用其中的代码也不是一件容易的事情，毕竟在我们的语法分析器中引入一个组合子或帮助函数去抽象一个通用模式是不可能的。

在语法分析器组合子库中，语法分析器只是一些普通的 first-class 值。重用语法分析逻辑是一件再平常不过的事情，以至于我们不需要任何外部的工具在语言之外做一些事情。

9.1 代数设计，走起

回忆之前我们定义的*代数*，它是一组基于某些数据类型的函数操作，以及这些函数间的法则。之前的章节中，设计的过程就是在创造函数、精炼函数，以及微调数据类型间往复。法则是在类型清晰，API 成型之后才产生的。这种设计风格没什么错，[1] 只是我们还可以有别的方式，比如先定代数（包括其法则）后定类型。这种方式就叫*代数设计*，它适用于任何设计问题，尤其是语法分析上，因为它能让你非常直观地知道什么样的输入需要什么样的组合子来进行分析。[2] 即便是延迟定义类型，我们也能专注在具体的目标上。

尽管已经存在很多不同种类的语法分析库。[3] 但我们的设计会表达力更强（能够分析任何语法），速度更快，错误提示更友好。最重要的一点是，无论何时运行分析器，只要输入不满足预期，像某些稀奇古怪的输入，它就会提示错误信息。一旦出现错误，我们希望能够指出错误的具体位置，并能准确地说明错误原因。一般而言错误报告是分析库最后考虑的事情，但我们却会花很多的工夫在上面。

好吧，让我们开始吧。为了简单和快速，首先库所创建的分析器将会基于字符串的输入进行分析。[4] 其次要选一些分析任务帮助发现一个不错的代数描述。那么我们该选哪些任务呢？ Email 地址、JSON，还是 HTML？都不是，这些都可以放到后面再说。解析像 “abracadabra” 和 “abba” 这样重复乱序的字母组合，会是一个很好且简单的切入点。这听上去很傻，但此前我们见到过，一个简单的例子是怎么帮助我们忽略无关细节，并关注问题本身的。

所以，我们就以它开始创建最简单的，一个仅识别一个字符 `'a'` 的分析器吧。正如之前章节那样，*创造*（invent）基于它的组合子，`char`：

```
def char(c: Char): Parser[Char]
```

1 关于更多不同的函数设计方法，请见本章的章节笔记。

2 我们将会看到用于语法分析的代数与各类计算机科学研究的语言（普通的、上下文无关的、上下文相关的）之间的联系。

3 Scala 标准库就提供了一个语法分析器库。在之前的章节里我们就曾强调，编写我们自己的库首先是为了教学，其次倡议没有库是绝对权威的。标准库中的实现在执行速度和良好错误提示上都不能满足我们的要求（请看章节笔记中的额外讨论）。

4 这是最简单的设计选择，我们可以做到更通用，但会付出一些代价。详见章节笔记中的讨论。

发生了什么？我们就这样变出了一个类型，Parser，它还有一个类型参数，用以表明Parser的结果类型（result type）。是的，要知道运行一个分析器不是返回一个是或否的结果那么简单，而是一旦成功，我们希望是一个有类型的结果；一旦失败，我们也能获知失败的相关信息。具体来讲，就是char('a')这个分析器有且仅当输入是字符'a'时，才会返回一个和'a'一样的结果。

既然说到了“运行分析器”，那总要有个办法实现它。这里就为它创造另一个函数：

```
def run[A](p: Parse[A])(input: String): Either[ParseError, A]
```

等等，ParseError是个什么鬼？这又是我们变出来的另一个类型！在这儿，其实我们没必要关心ParseError的表现，或Parser具体是什么，我们只需要知道它们不过是接口上用到的两个类型罢了，至于它们的实现细节暂时忽略掉。来看一个严格完成的trait定义：

```
trait Parsers[ParseError, Parser[+_]] { ←——Parser有一个协变类型构造器。
  def run[A](p: Parser[A])(input: String): Either[ParseError, A]
  def char(c: Char): Parser[Char] ←——这里Parser类型参数是Char。
}
```

Parser[+_]参数类型看上去是不是有点奇怪呢？细节现在还不重要，我们只需要了解它是一种类型参数的Scala语法，表示其自类型的构造器。[5]ParseError的类型参数让Parsers接口可以使用任何错误类型，而Parser[+_]也让Parsers接口可以使用任何类型的Parser。其中下画线意味着Parser需要一个结果类型，任何类型都行，就好像Parser[Char]，这样编译才能通过。这里不用明确ParseError和Parser的具体类型，后续我们还会往trait中添加其他类型的组合子。

对于任何Char c而言，函数char都应满足下面的法则：

```
run(char(c))(c.toString) == Right(c)
```

在实现了识别单个字符'a'之后，我们如何做到识别字符串abracadabra呢？既然现在没有办法识别整个字符串，那我们不妨添加一个函数：

```
def string(s: String): Parser[String]
```

同样，对于任何String s而言，它也应该满足：

```
run(string(s))(s) == Right(s)
```

那如果我们要实现识别两个字符串中任意一个呢？如"abra"或"cadabra"。我们可以为此添加一个特定的组合子：

```
def orString(s1: String, s2: String): Parser[String]
```

这看上去不太通用，要是能在两个分析器中进行选择看起来会更好。比方说这样：

```
def or[A](s1: Parser[A], s2: Parser[A]): Parser[A]
```

当or(string("abra"), string("cadabra"))时，我们则期望在它们俩任意一个出现后都应该分析成功：

```
run(or(string("abra"),string("cadabra")))("abra") == Right("abra")
run(or(string("abra"),string("cadabra")))("cadabra") == Right("cadabra")
```

5 更多相关内容还会在本章后面提及。

顺带的，我们可以为 or 组合子支持一种中缀语法写法，如 s1|s2，这等同于 s1 or s2。类似第 7 章的做法，我们用隐喻的方式实现：

示例 9.1　为 Parser 支持中缀语法

```
trait Parsers[ParseError, Parser[+_]] { self =>
    ...
    def or[A](s1: Parser[A], s2: Parser[A]): Parser[A]
    implicit def string(s: String): Parser[String]
    implicit def operators[A](p: Parser[A]) = ParserOps[A](p)
    implicit def asStringParser[A](a: A)(implicit f: A =>
      Parser[String]): ParserOps[String] = ParserOps(f(a))

    case class ParserOps[A](p: Parser[A]) {
      def |[B>:A](p2: Parser[B]): Parser[B] = self.or(p,p2)
      def or[B>:A](p2: => Parser[B]): Parser[B] = self.or(p,p2)
    }
}
```

这里的self是Parsers实例的引用命名，它在后面的ParserOps中会用到。

使用self来消除or方面的调用二义性。

这里我们还将 string 声明为作为隐式转换，并提供了 asStringParser。有了这两个隐式函数，Scala 会自动将 String 转换成一个 Parser，这样一来对于任何可转换为 Parser[String] 的类型都将支持中缀操作符。即，若给定 val P: Parsers，在 import P._ 之后，我们便可以用 "abra" | "cadabra" 来创建分析器了。这对于所有 Parsers 的实现都是适用的。其他类似的二元操作符或方法都会添加到 ParserOps 上，而保持主要的定义直接声明在 Parsers 上，尽管最终还是代理给了 ParserOps。本章余下的部分，我们会大量用到 a | b 这样的语法，来表示 or(a, b)，因此类似的上面的代码还会出现在很多例子中。

现在我们可以识别各种字符串了，但还不能支持重复。比方说，如何识别重复了三次的 "abra" | "cadabra"？那么就再加个组合子吧：[6]

```
def listOfN[A](n: Int, p: Parser[A]): Parser[List[A]]
```

之所以为 listOfN 定义了类型参数 A，那是因为我们并不关心传入的是 Parser[String]，或是 Parser[Int]，还是其他什么类型。这里我们关心的是它能够满足下面的期望：

```
run(listOfN(3, "ab" | "cad"))("ababcad") == Right("ababcad")
run(listOfN(3, "ab" | "cad"))("cadabab") == Right("cadabab")
run(listOfN(3, "ab" | "cad"))("ababab") == Right("ababab")
```

至此，我们根据需要定义了这些组合子，但还没有尝试精炼成最小的原语集，也没有提及相关的法则。这并不代表我们放弃不做了，相反下一节中我们就会开始做它们。而接下来，我们希望你能借助一些简单的用例检验自己，并尝试设计出一个最小范围的代数及其法则。这会是个非常有挑战的练习，尽情享受其中的纠结和痛苦，也看看你自己到底能做到什么程度。

这里有一些额外的分析任务和一些引导问题供你思考：

6　这儿可能会让你想起上一章写过的一个类似的函数。

- 一个 `Parser[Int]` 能识别零或多个 `'a'` 字符，返回的结果是 `'a'` 出现的个数。比如，输入 `"aa"`，则分析的结果是 `2`；输入 `""` 或 `"b123"`（一个不是以 `'a'` 开头的字符串），则结果为 `0`；等等。
- 一个 `Parser[Int]` 能识别一个或多个 `'a'` 字符，返回的结果也是 `'a'` 出现的个数。（和上一个是不是很相似呢？）当输入的字符串不是 `'a'` 开头的则分析失败。这种情况下会如何提示错误呢？API 上能否支持针对这类情况的错误消息定制呢？比如 `"Expected one or more 'a'"`。
- 一个分析器能够识别零或多个 `'a'`，以及紧跟其后的一到多个 `'b'`，然后以元组方式返回分析到的个数。例如，输入 `"bbb"`，结果是 `(0, 3)`；输入 `"aaaab"`，结果是 `(4, 1)`；等等。

一些额外的考量：

- 如果我们尝试分析一系列零到多个 `"a"`，且只关心分析到的个数，要是用 `List[Char]` 可能显得不够高效，毕竟在用完后取了长度就扔掉了。有没有更有效的做法呢？
- 在我们的代数中是否必须要有多种形式的重复原语？或者说它们能否有更简单的表现形式？
- 很早我们就引入了 `ParserError`，但一直没有为它定义任何 API，以至于程序员无法控制提示什么错误，也就限制了分析器最终无法提供更友好的错误信息。关于此，你能做点什么改进吗？
- `a | b` 等价于 `b | a` 吗？这个你来决定。若是，将会导致什么结果？反之，又会如何？
- `a | (b | c)` 等价于 `(a | b) | c` 吗？若是的话，这是你的代数中原语级的法则吗？还是说有更简单的描述呢？
- 尝试总结一些法则，不必特别全面，只要你觉得能够应验到任何 `Parsers` 的实现即可。

花些时间去基于上面的指引总结出组合子和可能的法则，当你遇到困难了，就停下来继续下一节的阅读，其中的问题便会迎刃而解。

代数设计的好处

当你先设计库的代数时，代数的数据类型的形式就显得没那么重要啦。一旦它们支持必要的法则和函数，你甚至都无须暴露其中的数据类型形式。

有这么一个观点认为，类型的含义是由它与其他类型的关系决定的，而非其内在形式（internal representation）。[7] 这个视角常常与范畴论（category theory）相结合，

7 这样的视角与面向对象设计相结合也是可以的，只是 OO 不太强调代数法则。更重要的理由是，OO 的封装性不允许对象存在的可变状态被公开，以免调用代码（client code）来捣乱。而这一点 FP 确实完全不用担心。

范畴论是此前我们已经提到过的一个数学分支。有关这些内容的链接，若有兴趣可查看章节笔记。

9.2 一种可能的代数

接下来我们继续基于之前的例子去发现新的组合子，若是你自己设计的话，很有可能有些差异，没关系的。

首先，根据那个分析重复字符 'a' 返回发现字符个数的分析器，我们为此添加一个原语组合子，就叫它 many 好了：

```
def many[A](p: Parser[A]): Parser[List[A]]
```

这貌似不是我们想要的啊，我们要的是一个 Parser[Int] 能够返回读到的元素个数。别急，我们可以改变 many 组合子令其返回 Parser[Int]，但这明显太有针对性了，万一我们想知道个数之外的东西呢？比较好的做法是引入另一个组合子 map：

```
def map[A, B](a: Parser[A])(f: A => B): Parser[B]
```

如此一来就可以这样来返回个数了：

```
map(many(char('a')))(_.size)
```

把 map 和 many 添加到 ParserOps 上，写起来就顺眼了：

```
val numA: Parser[Int] = char('a').many.map(_.size)
```

那么，对于 run(numA)("aaa") 其结果是 Right(3)，而 run(numA)("b") 则为 Right(0)。

map 远不只是在 Parser 运行成功后转换其结果，还应该在没有额外输入字符时发现问题，也就是说一个失败的分析器经过 map 后不可能是一个成功的分析器，反之亦然。因此，一般而言 map 应具备结构保持（structure preserving）的特性，这一点与 Par、Gen 一致。用法则描述的话就是：

```
map(p)(a => a) == p
```

这个法则该如何文档化呢？我们可以在注释里描述，但这显然不如利用上一章我们开发的库写一个可以执行的法则来得要好：

示例 9.2 组合 Parser 与 map

```
import fpinscala.testing._

trait Parsers[ParseError, Parser[+_]] {
    ...
    object Laws {
        def equal[A](p1: Parser[A], p2: Parser[A])(in: Gen[String]): Prop =
            forAll(in)(s => run(p1)(s) == run(p2)(s))

        def mapLaw[A](p: Parser[A])(in: Gen[String]): Prop =
            equal(p, p.map(a => a))(in)
    }
}
```

这在验证 Parsers 实现满足预期上非常有帮助，后续更多的法则最好都用这个办法。[8]

顺带的，有了 map，我们其实可以用 string 来实现 char：

```
def char(c: Char): Parser[Char] =
  string(c.toString) map (_.charAt(0))
```

另一个类似的组合子 succeed，可以用 string 和 map 来实现：

```
def succeed[A](a: A): Parser[A] = string("") map (_ => a)
```

无论输入什么样的字符串，这个分析器总是成功地返回值 a（即便是空字符串，string("") 也会成功）。这个组合子看着是不是很眼熟啊？那它的行为法则是：

```
run(succeed(a))(s) == Right(a)
```

9.2.1 切片和非空重复

尽管借助 many 和 map 可以实现分析 'a' 字符计数的任务，但这种做法看上去并不高效，即构造一个 List[Char] 丢弃其元素不用仅获取长度。如果能做到运行一个 Parser 仅得到部分输入的字符串，就会好得多了。为此我们再变出一个组合子：

```
def slice[A](p: Parser[A]): Parser[String]
```

我们之所以叫这个组合子为 slice，是因为当分析器执行成功时仅返回了检查过的部分输入字符串。比方说，run(slice(('a' | 'b').many))("aaba") 的结果是 Right("aaba")，这里我们忽略了 many 积累的列表，只是简单地返回了分析器匹配到的部分输入字符串。

有了 slice，我们的分析器在计算 'a' 字符的个数时，可以写成 char('a').many.slice.map(_.size)（假定 ParserOps 上添加了 slice 方法）。那么 _.size 调用就是 String 的 size 方法，这个耗时是个常量，而不再是 List 的 size，那个时间的消耗就是随着列表的长度线性增长的（而且必须要构造这个列表）。

注意，现在我们还没有实现，这还只是一个设想中的接口而已。倒是 slice 为实现添加了一个约束，即运行 p.many.map(_.size) 是会理解构造列表的，而 slice(p.many).map(_.size) 就不会。其实这已经强烈地预示着 slice 是一个原语级的函数，且必须要了解分析器内部的表现形式。

我们再来看下一个用例，如果我们想要识别一到多个 'a' 字符呢？首先，我们引入一个新的组合子，叫 many1：

```
def many1[A](p: Parser[A]): Parser[List[A]]
```

感觉上 many1 并不需要是原语级的函数，而是基于 many 实现的，即 many1(p) 就是一个 p 后面接着一个 many(p)。那看起来我们需要一个方法能够成功执行完一个分析后，再执行另一个。

```
def product[A, B](p: Parser[A], p2: Parser[B]): Parser[(A, B)]
```

---

8 同样的，章节中的代码有更多的例子供参考。这里为了保持代码简短，就不提供所有法则的实现了，但这并不意味着你自己不用去写它们。

为 ParserOps 加上 product 和 ** 两个方法，让 a ** b 和 a product b 都代理到 product(a,b) 上。

◈ 练习 9.1

使用 product 实现类似的组合子 map2，然后用它再去组合 many 去实现 many1。注意，我们可以让 map2 成为原语级的函数，而让它去实现 product，就像上一章我们做过的那样。两种方式怎么选择由你来决定。

```
def map2[A, B, C](p: Parser[A], p2: Parser[B])(f: (A, B) => C): Parser[C]
```

有了 many1，我们就可以实现分析零到多个 'a'，并且紧跟一到多个 'b' 的情况了：

```
char('a').many.slice.map(_.size) ** char('b').many1.slice.map(_.size)
```

◈ 练习 9.2

难：尝试总结 product 的法则。

现在有了 map2，那 many 还是原语级的吗？仔细想想 many(p) 都干了些什么，先是运行了 p，然后跟着运行 many(p)，然后重复 many(p) 直到分析失败，最后返回期间积累的 p 的列表。如果一开始就失败，则返回空列表。

◇ 练习 9.3

难：在继续之前，看看你能否用 or、map2 还有 succeed 实现 many？

◇ 练习 9.4

难：使用 map2 和 succeed 实现 listOfN。

```
def listOfN[A](n: Int, p: Parser[A]): Parser[List[A]]
```

好，我们来尝试用 or、map2 还有 succeed 实现 many：

```
def many[A](p: Parser[A]): Parser[List[A]] = map2(p,many(p))(_ :: _) or
succeed(List())
```

看着还挺简约的。我们用 map2 实现了 p 到 many(p) 的串联，再将其结果用 :: 构造成列表。或者，开始失败就通过 succeed 返回空列表。可是这里有个问题，你看出来了吗？这里我们在 map2 的第二个参数上递归调用了 many，它是会被严格求值的。试想一下这个求值的过程：

```
many(p)
map2(p, many(p))(_ :: _)
map2(p, map2(p, many(p))(_ :: _))(_ :: _)
map2(p, map2(p, map2(p, many(p))(_ :: _))(_ :: _))(_ :: _)
...
```

由于对 map2 的调用总是会对第二个参数求值，那么 many 函数将永远不会停止！这就坏了。我们必须让 product 和 map2 对第二个参数进行惰性求值：

```
def product[A,B](p: Parser[A], p2: => Parser[B]): Parser[(A,B)]

def map2[A,B,C](p: Parser[A], p2: => Parser[B])(
                f: (A,B) => C): Parser[C] =
  product(p, p2) map (f.tupled)
```

◇ **练习 9.5**

我们还可以用单独的组合子实现非严格求值，在第 7 章中我们也这么干过。尝试一下不改变现有的组合子，实现这么一个组合子。

现在我们实现的 many 应该可以很好地工作了。理论上讲，只要 product 的第二个参数是惰性求值的，那么一旦第一个 Parser 失败，第二个参数就不会被求值。

现在我们实现的组合子都在成功的情况下，才进行串联执行。那我们再回头看看 or：

```
def or[A](p1: Parser[A], p2: Parser[A]): Parser[A]
```

我们假定 or 是左偏向的（left-biased），即在 p1 失败后才尝试运行 p2。[9] 这种情况下，毫无疑问第二个参数是需要改成非严格求值的：

```
def or[A](p1: Parser[A], p2: => Parser[A]): Parser[A]
```

9.3 处理上下文的相关性

目前已经浮现的原语级函数如下：

- string(s)——识别并返回一个字符串
- slice(p)——返回部分由 p 分析成功的字符串输入
- succeed(a)——总是成功地返回值 a
- map(p)(f)——在 p 执行成功后，其结果应用函数 f
- produce(p1, p2)——串化两个分析器，先执行 p1 后执行 p2，在二者皆成功时返回它们的结果
- or(p1, p2)——在两个分析器中选择执行，先尝试 p1，失败后再尝试 p2

使用它们，我们可以表达重复和非空重复（many、listOfN 和 many1），也可以是 char 和 map2 这样的组合子。就这些原语级的函数已经能够分析任何上下文无关的语法了，包括 JSON，对此你是否感到惊讶呢？是的，它们确实做得到！我们马上就会写一个 JSON 的分析器，只是还有一些东西是*无法*表达的，它们是什么呢？

9 这是一个设计选择。你可能会愿意思考一下，若是 or 总是执行 p1 和 p2 会是一个怎样的结果。

假设我们想要分析一个数字，像是 '4'，和之后的若干个字符 a（看起来似乎和上个章节的问题相似）。其中的一些例子是 "0"、"1a"、"2aa"、"4aaaa" 等，它们就是典型的上下文相关的语法。它们是无法用 produce 来表达的，因为第二分析器的选择取决于第一个分析的结果（即第二个分析器依赖其上下文）。就好比让 listOfN 是使用第一个分析器的分析得到的数字结果。现在你能明白 produce 为什么不能表达了吧？

进展到此又有了似曾相识的感觉。在此前章节里遇到类似的限制时，我们是通过引入新的原语 flatMap 来解决的。这里不妨如法炮制（为 ParserOps 添加别名，以便支持 for 推导的写法）：

```
def flatMap[A, B](p: Parser[A])(f: A => Parser[B]): Parser[B]
```

你能看出它是怎么实现下一个分析器依赖前一个分析器的吗？

◈ **练习 9.6**

使用 flatMap 和其他组合子，实现之前无法表达的上下文相关的分析器。实现一个新的原语 regex 能够分析数字，即让一个正则表达式变成一个 Parser。[10] 在 Scala 中，字符串 s 可以通过 s.r 变成一个 Regex 对象（它有匹配相关的方法），比如 "[a-zA-Z_][a-zA-Z0-9_]*".r。

```
implicit def regex(r: Regex): Parser[String]
```

◈ **练习 9.7**

使用 flatMap 实现 product 和 map2。

◈ **练习 9.8**

使用 flatMap 和（或）其他组合子实现 map。

有了这么一个新的原语 flatMap，我们可以实现上下文相关的分析，还可让我们实现 map 和 map2。反正这也不是第一次拿 flatMap 干这事了。

现在的原语集更小了，只有 6 个：string、regex、slice、succeed、or 和 flatMap，但表达力却更强。用 flatMap 替代 map 和 product，不仅能够分析上下文无关的语法，如 JSON；还可以分析上下相关的语法，包括极其复杂的 C++ 和 PERL！

9.4 写一个 JSON 分析器

我们这就动手写一个 JSON 分析器，如何？尽管我们的代数还未实现，也没有良好的错误提示，但这些都可以稍后再说。目前 JSON 分析器还无须了解其实现的细节，仅靠已有的原语和组合子即可简单地实现了。

10 理论上这是没有必要的；我可能可以写出 "0" | "1" | ... "9" 去识别一个数字，但这样似乎效率不高。

也就是说，对于某些 JSON 分析的结果类型（后文会简要谈及 JSON 格式和分析的结果类型），其函数会是这样的：

```
def jsonParser[Err, Parser[+_]](P: Parsers[Err, Parser]): Parser[JSON] = {
    import P._ ←——引入所有的组合子。
    val spaces = char(' ').many.slice
    ...
}
```

看起来可能会有点奇怪，毕竟在实现 Parsers 接口之前我们是没法实际运行分析器的。尽管如此，我们还是可以继续的，因为在函数式编程里，往往定义一个代数，即便是没有实现的情况下，也可以了解其表达力。相反，具体的实现会限制我们，使得 API 的变化很困难。尤其在库的设计阶段，没有具体的实现代码，反而更容易精炼我们的代数。另外，这样做的部分原因也是想让你逐渐适应这样的工作方式。

本节之后，我们会回到如何为此添加更好的错误提示的问题上来，并在不改变多少现有 API 结构和 JSON 分析器的前提下，完成一个具体的，可运行的 Parser 类型的形式（representation）。关键是，你会发现下一节中实现的 JSON 分析器是完全独立 Parser 类型的形式。

9.4.1 JSON 格式

如果你还不太熟悉 JSON 的格式，那可能需要阅读一下 Wikipedia 上的描述（http://mng.bz/DpNA）和语法说明（http://json.org）。下面是一段 JSON 文档示例：

```
{
    "Company name"       : "Microsoft Corporation",
    "Ticker"             : "MSFT",
    "Active"             : true,
    "Price"              : 30.66,
    "Shares outstanding" : 8.38e9,
    "Related companies"  : [ "HPQ", "IBM", "YHOO", "DELL", "GOOG" ]
}
```

JSON 中值的类型可能是下面几个类型之一。一个对象就是一对花括弧（{}）包起来的，且用逗号分隔的键值序列。其中键必须是像 "Ticker" 和 Price 的字符串，而值的话要么是对象，要么是 ["HPQ", "IBM"...] 这样的数组，再就是 MSFT、true、null 或 30.66 这样的常量（literal）。

我们将实现一个最简单的分析器，除了从 JSON 文档中分析语法树外什么也不做。[11] 但在此之前我们需要一个分析后的 JSON 数据类型：

```
trait JSON
object JSON {
    case object JNull extends JSON
    case class JNumber(get: Double) extends JSON
    case class JString(get: String) extends JSON
```

11 查看章节备注有关其他方法的讨论。

```
    case class JBool(get: Boolean) extends JSON
    case class JArray(get: IndexedSeq[JSON]) extends JSON
    case class JObject(get: Map[String, JSON]) extends JSON
}
```

9.4.2 JSON 分析器

回顾一下现在已有的原语：

- `string(s)`：识别并返回一个 `String`
- `regex(s)`：识别一个正则表达式 `s`
- `slice(p)`：返回 `p` 分析成功后的输入字符串
- `succeed(a)`：始终返回值 `a`
- `flatMap(p)(f)`：执行一个分析器，然后根据它的结果选择第二个分析器继续执行
- `or(p1, p2)`：在两个分析器中选择执行，先尝试 `p1`，若失败后执行 `p2`

还有基于这些原语实现的计算组合子，像 `map`、`map2`、`many` 和 `many1`。

◈ **练习 9.9**

难：现在要由你来接手，借助我们定义的原语创建一个 `Parser[JSON]`。无须关心 `Parser` 的类型表现。过程中，你无疑会发现新的组合子或习惯用法，注意并提取其中的通用模式。使用从本书中学到的技巧，好好享受这个过程吧。在你遇到困难的时候再看答案。下面是一些最小化的建议：

- 你发现的任何通用组合子都可以直接添加到 `Parsers` 上。
- 为了让分析词法（像字符串和数字）更容易，你可能需要引入新的组合子。可以用 `regex` 来实现，也可以使用新的原语，如 `letter`、`digit`、`whitespace` 等。

一个完整的 JSON 分析器实现在文件 `JSON.scala` 中，在你需要更多帮助的时候可以参考。

9.5 错误提示

直到现在我们都还完全没有讨论过错误提示，而是聚焦在发现各种原语上，以便使分析器能够表达各种语法。可是除了分析语法之外，我们还是希望分析器能够对意外的输入有合理的反馈。

即便是在不知道 `Parsers` 的实现细节的情况下，我们依旧可以大致推理出这些组合子应该提供怎样的信息。哪怕是这些组合子都未曾在失败的时候提及任何关于错误消息是什么，或是 `ParserError` 具体包含什么，却只定义了语法是什么和成功后输出的结果。若

我们认为这样就算好了，然后开始具体的实现，则必定会在错误提示和错误消息上做出草率的决定，这不是我们的常规做法。

◇ **练习 9.10**

难：若你尚未开始，那就花些时间去发现一组不错的组合子能为 Parser 提示错误。对于每个组合子，去尝试用法则描述它应有的行为。这是一个非常开放的设计练习，这里依旧给你一些建议：

- 假定分析器为 "abra".**(" ".many).**("cadabra")，当输入 "abracAdabra"（注意其中大写的 'A'）时，那应该提示什么错误？是 Expected 'a'，或是 Expected "cadabra"，还是完全不同的消息，如 "Magic word incorrect, try again!"？
- 对于 a or b，若 a 失败，我们总是要运行 b 吗，还是某些情况下希望不运行呢？若是存在这样的情况，请思考一下有没有一个组合子可以让程序员指定 or 要不要运行第二个参数？
- 你打算如何报告错误的位置？
- 对于 a or b，当二者都失败时，我们会想要提示二者的错误吗？是一直提示二者的错误呢，还是让程序员有办法指定提示是哪个的错误呢？

我们建议你的设计令自己满意了再继续阅读，下一节我们将讨论一个可行的设计细节。

组合子的特定信息

在典型的库设计场景中，我们至少对库的具体形式有些初步的了解，并常常基于它对形式的影响来考量函数。而自从采用了代数优先后，我们不得不换个思路，即以它能提供给实现的信息来考量函数。函数的签名就提供了这样的信息，其实现可随意使用这些信息只要满足任何特定的法则。

9.5.1 一种可行的设计

现在你已经在设计错误提示的组合子上花了一些时间了，接下来我们就要一起来讨论一个可行的设计。还是那句话，你的设计可能与之完全不同，但这没有任何影响。接下来的只是另一种设计的形成过程而已，label：

```
def label[A](msg: String)(p: Parser[A]): Parser[A]
```

label 的意义在于，当 p 失败后，其 ParseError 将会收录消息 msg。这意味着什么？意味着我们可以假定 type ParserError = String，且返回的 ParseError 就等同于 label。可是，我们还希望错误发生时能够告诉我们问题发生的位置。那不如暂时这样：

```
case class Location(input: String, offset: Int = 0) {
  lazy val line = input.slice(0,offset+1).count(_ == '\n') + 1
  lazy val col = input.slice(0,offset+1).lastIndexOf('\n') match {
```

```
        case -1 => offset + 1
        case lineStart => offset - lineStart
    }
}

def errorLocation(e: ParseError): Location
def errorMessage(e: ParseError): String
```

Location 中包含了完整的输入，和当前输入的偏移量，用这两者便可以计算行列的数值。为此我们能从 label 提供更多的信息。倘若失败的结果是 Left(e)，errorMessage(e) 将得到 label 中设置的消息。可以用 Prop 来描述：

```
def labelLaw[A](p: Parser[A], inputs: SGen[String]): Prop =
    forAll(inputs ** Gen.string) { case (input, msg) =>
        run(label(msg)(p))(input) match {
            case Left(e) => errorMessage(e) == msg
            case _ => true
        }
    }
```

Location 在哪里？它会在 Parsers 的实现中被填入错误发现的地方。这听上去还是有点模糊，如果 a or b 都失败了，这要提示哪个位置，哪个 label 呢？我们将在下一节讨论。

9.5.2 错误嵌套

label 组合子就能满足所有报告错误的需求了？不尽然，我们看个例子：

```
val p = label("first magic word")("abra") **
        " ".many ** ←——跳过空格。
        label("second magic word")("cadabra")
```

我们想要从 run(p)("abra cAabra") 中得到一个怎样的 ParseError 呢？（注意 cAdabra 中大写的 A）。直接的原因就是应该是 'a' 却读到了 'A'。错误的位置信息最好也能显示出来。可这样的提示太底层，不够友好，尤其是在有大量语法结构，且在分析大量输入的情况下。要是能够有更多上下文的信息，如 "second magic word"，会好很多。理想情况下，错误信息应该告诉我们在分析 "second magic word" 的时候，意外地出现了大写的 'A'。这不仅准确地指明了错误发生的地方，还提供了易于理解的上下文。又或者这里的顶级分析器（这里的 p）可以在失败时，提供关于分析的更高层的描述（比如说，"paring magic spell"），这同样很有帮助。

因此，这样看来仅仅提供一个层面的错误提示是不够的，我们要提供一种嵌套的 labels：

```
def scope[A](msg: String)(p: Parser[A]): Parser[A]
```

不像 label，scope 不会将 label(s) 抛给 p，它仅仅只是在 p 失败时添加额外的信息。让我们解释得更具体一点。首先，我们修改从 ParseError 中获取信息的函数，与其在其中获取一个 Location 和 String 消息，不如得到一个 List[(Location, String)]：

```
case class ParseError(stack: List[(Location,String)])
```

这是一个错误消息的栈，当 Parser 失败时能够说明它在做什么。现在我们就可以说清楚 scope 能做什么啦——如果 run(p)(s) 的结果是 Left(e1)，然后执行 run(scope(msg)(p))(s) 结果是 Left(e2)，此时 e2.stack.head 是 msg，而 e2.stack.tail 是 e1。

我们后续可以写一些帮助函数，使得构建和操作 ParseError 更容易一些，并格式化其内容使其更易读。但现在我们仅需要它能够保存所有错误提示的信息，而现在的 ParseError 看起来已经达到这个目的了。就让它成为具体的表现形式，且去掉 Parsers 的抽象类型参数：

```
trait Parsers[Parser[+_]] {
    def run[A](p: Parser[A])(input: String): Either[ParseError,A] ...
}
```

现在我们已经提供了构建 Parsers 实现的所需全部信息、可供选择的多层错误结构。作为 Parsers 库的使用者，我们可以有见地地用 scope 和 label 描述我们的语法，从而在 Parsers 实现中用于构建分析的异常。注意，已有充足的理由说明，Parsers 的实现与 ParseError 无关，它仅保存了错误原因和位置的基本信息。

9.5.3 控制分支和回溯轨迹

只剩下最后一个错误提升相关的问题需要我们讨论了。正如我们之前提及的，当 or 组合子遇到错误时，我们需要一种方法来决定提示哪些错误。我们不想只有一个全局的约定，而是能够让程序员控制这个选择。现在就来看一个具体的例子：

```
val spaces = " ".many
val p1 = scope("magic spell") {
    "abra" ** spaces ** "cadabra"
}
val p2 = scope("gibberish") {
    "abba" ** spaces ** "babba"
}
val p = p1 or p2
```

当从 run(p)("abra cAdabra") 返回后，ParseError 会是怎样的呢？（再次强调，cAdabra 中那个大写的 A）。针对这样的输入，两种情况都会出错。标记了 "gibberish" 的分析器会在期望读到 "abba" 时报错，而 "magic spell" 的分析器会因为 "cAdabra" 的大小写问题报错。到底哪些错误是我们希望提示给用户的呢？

在本例中，我们碰巧想要 "magic spell" 的分析错误，即在成功分析了 "abra" 之后，我们已经执行了 "magic spell" 分支，这就意味着一旦我们遇到了异常，就不用再去检查另一个分支了。而别的情况里，我们可能需要分析器去考虑另一个分支。

因此，我们需要一个原语来允许程序员选择提交哪条分析的分支。起初我们定义 p1 or p2 是对输入先尝试 p1，若 p1 失败再执行 p2，现在可以做些调整，对于输入先尝试

p1，若 p1 失败进入非提交状态，则继续执行 p2；否则，提示失败。这比仅提供良好的错误消息更有用，让实现避免检查一堆可能的分析分支。

一个常见的解决方案是让所有分析默认都被提交，只要它检查了至少一个字符且产生结果。[12] 然后我们引入一个组合子 attempt，它可以延迟提交：

```
def attempt[A](p: Parser[A]): Parser[A]
```

它应该能满足下面的表达式：[13]

```
attempt(p flatMap (_ => fail)) or p2 == p2
```

这里的 fail 是一个始终会失败的分析器（我们可以将它作为原语组合子引入进来）。这就使得 p 在中途失败时，attempt 令其回到初始的状态，并去执行 p2。每当语法中存在多种可能，需要尝试所有情况，才能确定最后的分支时，我们都可以使用 attempt 组合子。我们看一个例子：

```
(attempt("abra" ** spaces ** "abra") ** "cadabra") or ("abra" ** spaces "cadabra!")
```

假定输入是 "abra cadabra"，在分析了第一个 "abra" 后，我们不知道后面的是另一个 "abra"（第一个分支），还是 "cadabra"（第二个分支）。通过用 attempt 包装 "abra" ** spaces ** "abra"，则允许尝试第二个分支，除非分析到了第二个 "abra"，到那时才决定提交哪个分支。

◈ **练习 9.11**

你能想到其他能够让程序员决定 or 如何提示错误的原语函数吗？

注意到了吧，我们依旧还没有实现我们的代数！但是这个练习已经足够说明我们的组合子为库的使用者提供一种方法去表达正确信息给到最终的实现。如何使用这些信息并满足我们要求的法则，就要留给实现决定了。

9.6 实现代数

到现在，我们不断充实我们的代数，也基于它定义 Parser[JSON]。[14] 你是不是等不及要探个究竟呢？

我们还是先回顾一下吧：

- string(s)——识别并返回一个 String。
- regex(s)——识别一个正则表达式 s。

12 更多信息参见章节笔记。

13 其实还不止这些，即便在尝试分析器失败后执行 p2，我们还要在 p2 失败时提示两个分支相关的错误信息。

14 你可能希望再审视一下自定义的分析器，也就是上节中我们用来实现错误提示组合子的那些分析器。

- `slice(p)`——返回 `p` 分析成功后的输入字符串。
- `label(e)(p)`——当失败时，将 `e` 作为错误消息。
- `scope(e)(p)`——当失败时，将 `e` 添加到由 `p` 返回的错误栈上。
- `flatMap(p)(f)`——执行一个分析器，然后根据它的结果选择第二个分析器继续执行。
- `attempt(p)`——延迟提交到 `p` 直到它成功为止。
- `or(p1, p2)`——在两个分析器中选择执行，先尝试 `p1`，若失败后执行 `p2`。

◇ **练习 9.12**

难：在下一节中，我们将完成 `Parser` 的外形表现，并用它来实现 `Parsers` 接口。但在此之前，你倒是可以做些尝试。这是一个非常开放的设计任务，只是我们设计的代数对可能的实现有了很强的约束。你应该想出既简单又纯函数化的 `Parser` 外形表现，然后用它来实现 `Parsers` 接口。[15]

你的代码可能类似：

```
class MyParser[+A](...) { ... }

object MyParsers extends Parsers[MyParser] {
  // 各种原语
}
```

使用你定义的分析器类型替换掉 `MyParser`。在你遇到困难，或是需要帮助时，请继续阅读后文。

9.6.1 一种可能的实现

我们现在就来讨论 `Parsers` 的实现。我们的解析代数已经支持很多功能了，与其直接去定义 `Parser` 最终的外形表现，不如审视代数中的原语，推理其必要的信息，最终一步步地形成 `Parser` 的外形表现。

先从 `string` 组合子入手：

```
def string(s: String): Parser[A]
```

其中我们需要支持的一个函数是 `run`：

```
def run[A](p: Parser[A])(input: String): Either[ParseError,A]
```

光凭直觉，我们可以假定 `Parser` 就是一个 `run` 函数的简单外形表现：

```
type Parser[+A] = String => Either[ParseError,A]
```

利用这个我们可以实现 `string` 原语：

15 注意，当 `Parsers` 实现后，你尝试执行 JSON 分析器，很有可能会出现栈溢出错误。我们会在下一节的结尾讨论这个问题。

```
def string(s: String): Parser[A] =
    (input: String) =>
        if (input.startsWith(s))
            Right(s)
        else
            Left(Location(input).toError("Expected: " + s))
```

稍后我们会定义toError构建一个ParseError。

else 分支会构建一个 ParseError，若是直接构建会有点麻烦，为此我们在 Location 上引入一个帮助函数 toError：

```
def toError(msg: String): ParseError = ParseError(List((this, msg)))
```

9.6.2 串化分析器

到现在一切都还不错，我们有了 Parser 的外形表现，至少支持 string。不幸的是，要想表现 "abra" ** "cadabra" 这样的解析器，现有的外形表现是无法满足的。那是因为，一旦分析 "abra" 成功后，我们要考虑哪些是已经被消费的字符，而剩余哪些字符会继续被 "cadabra" 进行分析。为了能够支持这样的串化，我们要让 Parser 知道多少字符已经被分析过了，做到这一点还是蛮容易的：[16]

```
type Parser[+A] = Location => Result[A]
```

分析器返回一个 Result，它要么是成功，要么是失败。

```
trait Result[+A]
case class Success[+A](get: A, charsConsumed: Int) extends Result[A]
case class Failure(get: ParseError) extends Result[Nothing]
```

成功的时候返回已经分析的字符数量。

这里我们引入了新类型 Result，而不再使用 Either。当成功的时候，我们返回 A 类型的值以及已经消费的字符个数，以便更新 Location 的状态。[17] 这个类型是 Parser 的本质，一种允许失败的，类似我们在第 6 章构建的状态行为。它接收输入的状态，在成功的时候，返回一个值，和控制其状态如何扭转其他信息。

基于状态行为的理解，使得我们能够找到 Parser 的外形表现框架，让它支持各种花哨的组合子和我们规定的法则。那剩下就是简单分析每个原语需要跟踪的状态类型（Location 恐怕就不够用了），并且寻思一遍每个组合子是如何基于状态发生变化的细节。

◈ 练习 9.13

基于 Parser 的初始外形表现，实现 String、regex、succeed 以及 slice。其中 slice 的实现可能会很低效，毕竟要创建一个值然后就丢弃它。关于这个问题我们回头再讨论。

---

16 回忆一下，Location 包括了完整的输入字符串及其偏移量。

17 这里要是返回 (A, Location) 的话，将会让 Parser 有能力改变 Location 中的 input，这是我们不愿看到的。

9.6.3 标记分析器

再来看看 scope，当失败时，我们希望往 ParseError 栈里添加一个新的消息。让我们在 ParseError 上引入一个帮助函数来做这件事情，我们叫它 push：[18]

```
def push(loc: Location, msg: String): ParseError =
  copy(stack = (loc,msg) :: stack)
```

有了它，我们就可以实现 scope：

```
def scope[A](msg: String)(p: Parser[A]): Parser[A] =
  s => p(s).mapError(_.push(s.loc,msg))
```

在失败的时候，在错误栈上添加msg。

Result 上的函数 mapError，就是应用（apply）一个函数并将错误的情况传递给它：

```
def mapError(f: ParseError => ParseError): Result[A] = this match {
  case Failure(e) => Failure(f(e))
  case _ => this
}
```

由于我们在内部分析器返回后的压栈行为，这使得栈上有了更多的细节信息供后续的分析过程使用。举个例子，当 scope(msg1)(a ** scope(msg2)()b) 失败了，则栈上第一个错误是 msg1，紧跟着的是 a 产生的错误，然后是 msg2，最后是 b 产生的错误。

尽管可以用类似压栈的办法实现 label，但我们更愿意用 mapError 来实现：

```
def label[A](msg: String)(p: Parser[A]): Parser[A] =
  s => p(s).mapError(_.label(msg))
```

这里调用了ParseError上的一个帮助函数，名字也叫label。

我们在 ParseError 上增加了一个帮助函数，名字也叫 label。我们决定用 label 来修剪错误栈，即丢掉内嵌范围的细节信息，仅使用栈底最近的位置：

```
def label[A](s: String): ParseError =
  ParseError(latestLoc.map((_,s)).toList)

def latestLoc: Option[Location] =
  latest map (_._1)

def latest: Option[(Location,String)] =
  stack.lastOption
```

获取栈的最后一个元素，当栈为空时返回None。

◈ 练习 9.14

重新审视你的 string 实现，尝试用 scope 和（或）label 在错误发生时提供更有意义的错误消息。

---

18 copy 是 case class 自带的方法。它会创建一个对象的副本，只是其中某些属性被修改了。如果未指定任何需要改变的属性，则会创建一个与原来对象一模一样的副本。这个行为的背后，得益于 Scala 默认参数的机制。

9.6.4 故障转移和回溯

现在轮到 or 和 attempt 了。想想对于 or 而言我们期望它的行为是：先运行第一个分析器，若失败了则进入一个非提交的状态，再基于同样的输入运行第二个分析器。对于哪怕分析了一个字符也应该尽心提交情况，attempt(p) 可以将失败的提交变为非提交状态。

我们可以通过在 Failure 中添加一个 Boolean 值来说明是否处于提交的状态：

```
case class Failure(get: ParseError,
                   isCommitted: Boolean) extends Result[Nothing]
```

而 attempt 的实现就是取消任何失败的提交，我们可以在 Result 上定义一个帮助函数 uncommit 来实现：

```
def attempt[A](p: Parser[A]): Parser[A] = s => p(s).uncommit

def uncommit: Result[A] = this match {
  case Failure(e,true) => Failure(e,false)
  case _ => this
}
```

同时，or 的实现也根据 isCommitted 的值来决定是否要运行第二个分析器。即在分析器 x or y 中，当 x 成功，则整个都成功。当 x 失败，此前我们是提交错误且跳过了运行 y，而现在我们会让它进入一个非提交状态，运行 y 并忽略 x 的结果：

```
def or[A](x: Parser[A], y: => Parser[A]): Parser[A] =
  s => x(s) match {
    case Failure(e,false) => y(s)
    case r => r ←——失败则运行y，成功则跳过。
  }
```

9.6.5 上下文相关的分析

最后来看 flatMap，它允许上下文相关的分析，即第二个分析器的选择由第一个分析器的结果来决定。实现其实很简单，就是在调用第二个分析器之前更新位置，同样我们使用 Location 上的帮助函数 advanceBy。其中有个小细节，若第一个分析器消费了任何字符，我们要确保第二个分析器被提交，即使用 ParseError 上的帮助函数 addCommit。

示例 9.3 使用 addCommit 确保分析器被提交

```
def flatMap[A,B](f: Parser[A])(g: A => Parser[B]): Parser[B] =
  s => f(s) match {
    case Success(a,n) => g(a)(s.advanceBy(n)) ←——在调用第二个分析器之前更新源。
                         .addCommit(n != 0) ←——若有消费则提交。
                         .advanceSuccess(n) ←——成功后更新已消费的字符个数。
    case e@Failure(_,_) => e
  }
```

advanceBy 就是简单地增加偏移量：

```
def advanceBy(n: Int): Location = copy(offset = offset+n)
```

同样的，addCommit 也很简单：

```
def addCommit(isCommitted: Boolean): Result[A] = this match {
  case Failure(e,c) => Failure(e, c || isCommitted)
  case _ => this
}
```

最后，由 advanceSuccess 累加一下成功处理的字符个数。我们想要得到 flatMap 处理的字符个数，其实就是分析器 f 和分析器 g 处理的字符个数之和，也就是我们用 advanceSuccess 计算得来的：

```
def advanceSuccess(n: Int): Result[A] = this match {
  case Success(a,m) => Success(a,n+m)
  case _ => this ←——如果不成功，保持结果不变。
}
```

◈ **练习 9.15**

使用现在 Parser 的外形表现，实现余下的原语，包括 run，然后用各种各样的输入尝试运行 JSON 分析器。[19]

◈ **练习 9.16**

尝试一种更易读的方式来格式化 ParseError。尽管选择很多，但关键在于呈现的错误能够将消息与位置绑定在一起显示。

◇ **练习 9.17**

难：slice 组合子依然不够高效。比如说，many(char('a')).slice 还是会创建一个 List[Char]，然后再丢弃它。请思考如何调整一下 Parser 的外形表现来优化分片的实现？

◇ **练习 9.18**

or 组合子组合的分析器在运行时可能会丢失一些信息。当两个分析器都失败的时候，我们仅保留了第二个分析器的错误。要是我们想要显示所有的信息，或是从任何分支上获取最多的信息呢？请尝试改变 ParseError 的外形表现以便保留其他分支上的错误。

19 不幸的是，你会发现当遇到巨长的输入时（比如，[1,2,3,...10000]），会导致栈溢出。简单的解决方法显然是用 many 来避免使用栈来构建。这也意味着任何组合子都要通过 many 来限定以解决。请查看相关的解答内容讨论更为通用的解决方案。

9.7 小结

本章，我们引入了代数设计，一种编写组合子库的方法，我们使用它设计分析器的库，并基于此实现 JSON 分析器。过程中，我们发现了不少与之前章节类似的组合子，且有着相近的法则。在第三部分中，我们将会搞明白库与这些抽象通用结构之间的关联。

这已经是第二部分的最后一章，我们希望你学到了函数式设计的基础方法，更重要的是能启发你去动手设计任何你感兴趣的问题域的相关库。函数式设计不是专家才能拥有的能力，而是每一个程序员每天函数式相关工作的一部分。在开始第三部分之前，我们鼓励你去探索，尝试编写更多函数式的代码，设计一些自己的函数式库。享受这些纠结的过程，看看你都发现了什么。当你回来的时候，一些通用的模式和抽象就在第三部分等着你呢。

第三部分

函数设计的通用结构

我们已经使用函数式设计原理编写了一些库，并在第二部分使用这些原理解决了一些实际问题。此刻你应该能够很好地运用组合性（compositionality）和代数推导（algebraic reasoning）来解决工作中的编程问题。

第三部分将面向更宽泛的场景并借此发现函数式编程里的通用模式。那些在第二部分中用于解决真实问题的具体方法会被抽象成描述通用结构的理论。

这种抽象带来直接的益处是：消除重复代码。将这些抽象表达为类型、接口或函数，并在实际的程序中直接进行引用。更重要的是概念集成。在不同上下文的不同解法中识别出通用的结构，并为这些结构实例统一概念进而命名。随着经验的积累你会发现问题的共性。比如某天脱口而出“这个看起来像单子（monad）！”，那时的你已经很擅长于此了。除此之外使用统一的抽象名称也可以跟其他人进行高效的沟通。

第三部分不会像第二部分那样迂回曲折，相反将在每一章开始引入一个抽象的概念，给出定义，并关联到已经给出的例子。其主要目的是训练我们在编写库时识别出不同模式，并学会充分利用这些模式来编写代码。

10 Monoid

在第二部分结束时我们已经习惯了使用代数术语来描述数据类型，也就是数据类型支持哪些操作和哪些代数法则指导了这些操作。希望你注意到不同的数据类型的代数描述倾向于共用某些特定的模式。在这一章将开始定位这些模式和利用它们。

本章将首次介绍纯代数（purely algebraic）结构。先从一个简单的结构 monoid（幺半群）[1] 开始，它是一个代数定义。除了满足同样的代数法则外不同的 monoid 实例很少有关联，然而这种代数结构定义了实现实用的多态函数必需的所有法则。

选择从 monoid 开始是因为它简单、普遍存在且很实用。monoid 在编程中经常出现，操作列表、连接字符串或者在循环中累加都可以被解析成 monoid。接下来将看到 monoid 是如何在两个方面被使用的：将问题拆分成小部分然后并行计算和将简单的部分组装成复杂的计算。

10.1 什么是 monoid

现在来考虑拼接字符串的代数表达，`"foo" + "bar"` 得到 `"foobar"`，空串是这个操作的单位元（identity[2]）元素，也就是 `(s + "")` 或者 `("" + s)` 的值都是 `s`。进一步说，如果将三个字符串相加 `(r+s+t)`，由于这个操作是可结合的（associative），所以 `((r+s)+t)` 或 `(r+(s+t))` 的结果是一样的。

同样的规则也适用于整数相加，它也是可结合的，因为 `(x + y) + z` 等于 `x + (y + z)`，而且有一个单位元元素 `0`，当和其他整数相加时不会产生改变。同样乘法也一样，它的单位元元素是 `1`。

布尔操作符 `&&` 和 `||` 同样是可结合的，它们的单位元元素是 `true` 和 `false`。

---

1 monoid（幺半群）出自于数学理论，在 category 理论中它意味着某类的一个对象。这个数学联系虽然对了解本章节不是关键，但也可以在本章笔记中查看更多信息。

2 数学里也称为幺元。——译者注

这些只是一些简单的例子，但是像这样的代数随处可见，有一个术语描述此类代数是*monoid*。结合律（associativity）和同一律（identity[3]）法则一起被称作*monoid*法则。一个monoid有如下构成：

- 一个类型A。
- 一个可结合的二元操作op，它接收两个参数然后返回相同类型的值，对于任何x: A, y:A, z:A来说，这两种操作是等价的：op(op(x,y), z) == op(x, op(y,z))。
- 一个值，zero: A，它是一个单位元，对于任何x: A来说，zero与它的操作都等于x自身：op(x, zero) == x或op(zero, x) == x。

可以用Scala trait表达：

```
trait Monoid[A] {
  def op(a1:A, a2:A): A ←——满足op(op(x,y), z) == op(x, op(y,z))。
  def zero: A ←——满足op(x, zero) == x和op(zero, x) == x。
}
```

String monoid是这个trait的一个实例：

```
val stringMonoid = new Monoid[String] {
  def op(a1: String, a2: String) = a1 + a2
  def zero = ""
}
```

List的连接同样构建了一个monoid：

```
def listMonoid[A] = new Monoid[List[A]] {
  def op(a1: List[A], a2: List[A]) = a1 ++ a2
  val zero = Nil
}
```

代数结构的纯抽象

注意除了满足monoid法则，不同的monoid实例之间没有太多联系。什么是monoid？简单的答案是一个类型、一些操作和一些法则。一个monoid是一个代数结构，没有别的。当然，你可能通过考虑各种具体情况建立一些“直观感觉”，但是这些直觉是不准确的，而且不能保证所有的monoid都满足你的直觉。

◈ 练习10.1

给出monoid实例，用来处理整数相加、相乘和布尔操作。

```
val intAddition: Monoid[Int]
val intMultiplication: Monoid[Int]
val booleanOr: Monoid[Boolean]
val booleanAnd: Monoid[Boolean]
```

3 identity有多重意思，除了表示数学上的单位元或幺元，还有一致性、同一性等含义。——译者注

◈ **练习 10.2**

给出能够组合 Option 值的 monoid 实例。

```
def optionMonoid[A]: Monoid[Option[A]]
```

◈ **练习 10.3**

我们把参数和返回值是相同类型的函数称为自函数（endofunction）。[4] 为 endofunction 编写一个 monoid。

```
def endoMonoid[A]: Monoid[A => A]
```

◈ **练习 10.4**

使用第二部分开发的基于属性的测试框架为 monoid 法则实现一个属性，并使用这个属性去测试已经编写的 monoid。

```
def monoidLaws[A](m: Monoid[A], gen: Gen[A]): Prop
```

“有”一个 monoid 还是“是”一个 monoid

当程序员和数学家讨论一个类型是 monoid 和有一个 monoid 实例时，术语存在一些不一致。作为一个程序员，易于认为一个 `Monoid[A]` 的实例是一个 monoid，但是这是不准确的。monoid 实际上是类型和定义法则的实例。更准确的是类型 `A` 和 `Monoid[A]` 实例定义的操作构成了 monoid。不那么准确的话可以说“类型 `A` 是一个 monoid”或“类型 `A` 是 monoidal”。任何情况下，`Monoid[A]` 实例都是这个事实的简单证据。

这和我们说一个页面或屏幕构建了一个矩形或是一个矩形差不多。“是一个矩形”的说法不太精确（虽然仍然有些道理），但是“有一个矩形”的说法就比较奇怪了。

那么 monoid 是什么？简单地说是一个类型 A 和一个实现法则的 `Monoid[A]`。简短地说，一个 *monoid* 是一个类型，一个此类型的二元操作（满足结合律）和一个单位元元素（*zero*）。

这样定义的意义何在？和其他抽象的作用一样，通过 monoid 这个抽象提供的能力可以编写通用的代码，可以使用不知道具体类型的 monoid 去编写有趣的代码，在接下来的例子中将会看到。

10.2 使用 monoid 折叠列表

monoid 和列表有很紧密的联系。假如你仔细观察 `List` 中 `foldLeft` 和 `foldRight` 的签名，就会发现参数的类型很特别：

4 希腊语中前缀 endo 的意思是“在什么里”，所以 endofunction 的 codomain 是指在它的范围里。

```
def foldRight[B](z: B)(f: (A, B) => B): B
def foldLeft[B](z: B)(f: (B, A) => B): B
```

当 A 和 B 类型是一样时：

```
def foldRight(z: A)(f: (A, A) => A): A
def foldLeft(z: A)(f: (A, A) => A): A
```

monoid 的各个部分符合这些参数。如果有一个字符串的列表，可以传递 stringMonoid 中的 op 和 zero，用于将列表中的字符串拼接在一起。

```
scala> val words = List("Hic", "Est", "Index")
words: List[String] = List(Hic, Est, Index)

scala> val s = words.foldRight(stringMonoid.zero)(stringMonoid.op)
s: String = "HicEstIndex"

scala> val t = words.foldLeft(stringMonoid.zero)(stringMonoid.op)
t: String = "HicEstIndex"
```

注意在使用 monoid 折叠时 foldLeft 和 foldRight 的效果是一样的。[5] 这正是因为结合律（associativity）和同一律（identity）法则。左折叠就像左关联操作，右折叠和右关联操作类似，而 identity 元素可以在任意一边。

```
words.foldLeft("")(_ + _) == (("" + "Hic") + "Est") + "Index"

words.foldRight("")(_ + _) == "Hic" + ("Est" + ("Index" + ""))
```

可以编写一个通用的 concatenate 函数，使用 monoid 去折叠列表：

```
def concatenate[A](as: List[A], m: Monoid[A]): A = as.foldLeft(m.zero)(m.op)
```

但是假如列表中的元素类型不是 Monoid 实例，该怎么办？总是可以将列表 map 成另外的类型：

```
def foldMap[A,B](as: List[A], m: Monoid[B])(f: A => B): B
```

◈ **练习 10.5**

实现 foldMap。

◈ **练习 10.6**

难：foldMap 可以使用 foldLeft 或 foldRight 实现，但你也可以使用 foldMap 来实现 foldLeft 和 foldRight，试一试！

10.3 结合律和并行化

monoid 操作的结合律意味着可以自由选择如何进行数据结构的折叠操作，这就像列表操作那样。已经展示了使用列表的 foldLeft 或 foldRight 去调用满足结合律的函数，

5 foldLeft 和 foldRight 都有尾递归的实现。

对列表按照顺序向左或向右 reduce。假如有个 monoid，甚至可以使用*平衡折叠法*（balanced fold）对列表进行 reduce，这样对某些操作可能更有效率或易于并行化。

下面的例子中，假设有一个顺序集 a, b, c, d，我们对其进行折叠，右折叠如下所示：

```
op(a, op(b, op(c, d)))
```

左折叠如下所示：

```
op(op(op(a, b), c), d)
```

平衡折叠如下所示：

```
op(op(a, b), op(c, d))
```

平衡折叠允许并行化计算，因为两个内部的 op 调用是独立的，其计算可以同时运行。除此之外，当每个 op 的时间花费与参数的长度成正比时平衡树的结构可以变得更高效。比如，下面的表达式运行时的性能：

```
List("lorem", "ipsum", "dolor", "sit").foldLeft("")(_ + _)
```

每一次折叠，分配一个临时的字符串然后丢弃，下次又需要分配一个更大的字符串。字符串的值是不变的，当 a+b 时，需要分配一个字符数组然后将 a 和 b 的值复制到这个新数组。这个时间花费与 a.lenght+b.length 是成正比的。

下面是之前表达式的求值轨迹：

```
List("lorem", ipsum", "dolor", "sit").foldLeft("")(_ + _)
List("ipsum", "dolor", "sit").foldLeft("lorem")(_ + _)
List("dolor", "sit").foldLeft("loremipsum")(_ + _)
List("sit").foldLeft("loremipsumdolor")(_ + _)
List().foldLeft("loremipsumdolorsit")(_ + _)
"loremipsumdolorsit"
```

临时字符串被创建后立即被丢弃。相比更高效的方式是对半组合顺序集，这就是所谓的*平衡折叠*，首先构建 "loremipsum" 和 "dolorsit"，然后将它们加在一起。

◈ 练习 10.7

为 IndexedSeq 实现 foldMap，[6] 使用的策略是将顺序集拆分成两部分，递归地处理每一部分，然后使用 monoid 将结果加起来。

```
def foldMapV[A,B](v: IndexedSeq[A], m: Monoid[B])(f: A => B): B
```

◇ 练习 10.8

难：使用第 7 章中开发的库实现*并行*版的 foldMap。提示：实现一个组合子 par，它可以将 Monoid[A] 提升为 Monoid[Par[A]]，[7] 然后使用它实现 parFoldMap。

```
import fpinscala.parallelism.Nonblocking._
```

---

6 IndexedSeq 是 immutable 数据结构实现的一个接口，用于支持高效的随机访问，同样接口也包含了高效的 splitAt 和 length 方法。

7 将 Monoid 提升到 Par 上下文的能力会在第 11 章和第 12 章中进行更广泛的讨论。

```
def par[A](m: Monoid[A]): Monoid[Par[A]]
def parFoldMap[A,B](v: IndexedSeq[A], m: Monoid[B])(f: A => B): Par[B]
```

◈ 练习 10.9

难：使用 foldMap 来判断给定的 IndexedSeq[Int] 是否是有序的。需要想一个创造性的 Monoid。

10.4 例子：并行解析

举个有意义的例子，计算字符串里的单词个数。这是十分简单的解析问题。可以按顺序扫描字符串，寻找空格然后对连续的非空格的字符计数。像这样按顺序解析，解析器的状态可以简单表达成最后一个字符是否是空格。

但是现在处理的不仅仅是一个短字符串，而是巨大的文本文件，可能大到单台机器的内存放不下。如果可以对文件的各个部分进行并行计算将会非常好。策略是将文件分拆成多个可管理的块（chunk），并行处理这些块，最后将结果合并起来。这种情况下，解析器的状态可能会复杂一些，需要可以合并中间结果并且无论这个部分是在文件的开头、中间还是结尾。换言之，需要合并操作是可结合的。

为了保持其简单性，考虑一个简单字符串并假装是大文件：

```
"lorem ipsum dolor sit amet, "
```

假如对半拆分字符串，这可能会将一个单词拆分。在这个例子中，将产生 "lorem ipsum do" 和 "lor sit amet, "。当累加这些字符串的计算结果时需要避免重复计入同一个单词。所以这里仅仅将单词作为整型计数是不充分的。需要设计一个数据结构能够处理部分结果，比如半个单词 do 和 lor，以及记录完整的单词，比如 ipsum、sit 和 amet。

单词计数的部分结果可以表达成一个代数数据结构：

```
sealed trait WC
case class Stub(chars: String) extends WC
case class Part(lStub: String, words: Int, rStub: String) extends WC
```

Stub 是最简单的 case，表示没有看到任何完整的单词。Part 保存看到的完整单词的个数，lStub 保存左边的部分单词，rStub 保存右边的部分单词。

比如，对字符串 "lorem ipsum do" 计数的结果是 Part("lorem", 1, "do")，因为有一个确定的完整单词 "ipsum"。因为 lorem 左边和 do 的右边都没有空格，我们不能确定它们是完整的单词，所以不计数。对 "lor sit amet, " 的计数结果是 Part("lor", 2, "")。

◈ 练习 10.10

为 WC 编写 monoid 实例，并确保满足 monoid 法则。

```
val wcMonoid: Monoid[WC]
```

◈ 练习 10.11

使用 WC monoid 实现函数，用来递归拆分 String 并计算各自包含的单词个数，最后再汇总。

monoid 同态

假如你能察觉到一些法则，将发现在 monoid 的函数之间有个法则。比如字符串的连接 monoid 和整数累加 monoid。假如取两个字符串的长度相加，等于连接两个字符串后取其长度：

```
"foo".length + "bar".length == ("foo" + "bar").length
```

这里，`length` 是一个函数，它将 `String` 转化为 `Int` 并保存 monoid 结构。这样的函数称之为 monoid 同态（homomorphism）。[8] 一个 monoid 同态 f 定义为在 monoid `M` 和 `N` 之间对所有的值及 `x` 和 `y` 都遵守以下法则：

```
M.op(f(x), f(y)) == f(N.op(x, y))
```

同样的法则也存在于之前从 `String` 到 `WC` 的练习中。

当设计自己的库时这个特质十分有用。假如两个类型是 monoid 并且它们之间存在函数，好的做法是考虑这个函数是否可以保持 monoid 结构，并且测试其是否为 monoid 同态。

某些时候两个 monoid 之间是双向同态（homomorphism）的。monoid 同质（isomorphic，以 iso 为前缀的词根表示相等）是在 `M` 和 `N` 之间存在两个同态的函数 `f` 和 `g`，而且 `f andThen g` 和 `g andThen f` 是等同的函数。

比如，`String` 和 `List[Char]` monoid 的连接操作是同质的。两个 `Boolean` monoid `(false, ||)` 和 `(true, &&)` 通过取反（`!`）同样是同质的。

10.5 可折叠数据结构

在第 3 章中实现了 `List` 和 `Tree` 的数据结构，两者都是可以折叠的。在第 5 章中实现了 `Stream`，一个像 `List` 一样可以折叠的延迟结构。现在为 `IndexedSeq` 实现一个折叠函数。

当编写代码去处理这类结构中的数据时，通常不在意具体结构是什么（不管是 List 或 Tree），也不在意是否是延时或者提供有效的随机读写，等等。

比如，有个结构中是整型，需要计算它们的总和，可以使用 `foldRight`：

```
ints.foldRight(0)(_ + _)
```

8 homomorphism 源自于希腊语，homo 意味着“相同”，morphe 意味着“形状”。

以上代码片段，不需要关心 ints 的具体结构类型，它可以是 Vector、Stream 或者 List，或者任何一个包含 foldRight 方法的类型。把这种通用性表达成下面的 trait：

```
trait Foldable[F[_]] {
    def foldRight[A,B](as: F[A])(z: B)(f: (A,B) => B): B
    def foldLeft[A,B](as: F[A])(z: B)(f: (B,A) => B): B
    def foldMap[A,B](as: F[A])(f: A => B)(mb: Monoid[B]): B
    def concatenate[A](as: F[A])(m: Monoid[A]): A =
        foldLeft(as)(m.zero)(m.op)
}
```

这里抽象出一个类型构造器 F，就像在之前章节中构建的 Parser 类型。它表达为 F[_]，这里下画线表示 F 不是一个类型而是一个*类型构造器*，它接收一个类型参数。就像接收别的函数作为参数的函数称为高阶函数，Foldable 是*高阶类型构造函数*或者高阶类型（higher-kinded type）。[9]

◈ **练习 10.12**

实现 Foldable[List]、Foldable[IndexedSeq] 和 Foldable[Stream]。foldRight、foldLeft 和 foldMap 之间可以互相实现，但可能不是最有效的实现。

◈ **练习 10.13**

第 3 章提及了二叉树，为它实现 Foldable 实例。

```
sealed trait Tree[+A]
case object Leaf[A](value: A) extends Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]
```

◈ **练习 10.14**

编写 Foldable[Option] 实例。

◈ **练习 10.15**

编写通用的转化方法，将 Foldable 结构转化为 List：

```
def toList[A](fa: F[A]): List[A]
```

10.6 组合 monoid

Monoid 本身的抽象还不是那么引人入胜，在泛化 foldMap 的帮助下它也只是变得有趣一些。它真正的强大之处来自于它的组合。

9 就像值和函数有类型，类型和类型构造器会有种类。Scala 使用种类去跟踪一个类型构造器有多少类型参数，无论这些参数是协变的还是逆变的，同样也跟踪这些参数是什么种类。

这意味着，比如，假如类型A和B是monoid，那么tuple类型(A, B)同样也是monoid(称为*product*)。

◈ 练习 10.16

证明：当且仅有A.op和B.op的实现是可结合的(associative)，那么如下的op实现也是可结合的。

```
def productMonoid[A,B](A: Monoid[A], B: Monoid[B]): Monoid[(A,B)]
```

10.6.1 组装更加复杂的monoid

只需要包含的元素是monoid，某些数据结构就能构建成monoid。比如，当value类型是monoid时合并key-value Map的操作就能够构建monoid。

示例 10.1 合并key-value Map

```
def mapMergeMonoid[K,V](V: Monoid[V]): Monoid[Map[K, V]] =
    new Monoid[Map[K, V]] {
        def zero = Map[K,V]()
        def op(a: Map[K, V], b: Map[K, V]) =
            (a.keySet ++ b.keySet).foldLeft(zero) { (acc,k) =>
                acc.updated(k, V.op(a.getOrElse(k, V.zero),
                                    b.getOrElse(k, V.zero)))
            }
    }
```

使用这个简单的组合子(combinator)就可以组装出复杂的monoid：

```
scala> val M: Monoid[Map[String, Map[String, Int]]] =
     | mapMergeMonoid(mapMergeMonoid(intAddition))
M: Monoid[Map[String, Map[String, Int]]] = $anon$1@21dfac82
```

这允许使用monoid而不需要额外构建嵌入表达式：

```
scala> val m1 = Map("o1" -> Map("i1" -> 1, "i2" -> 2))
m1: Map[String,Map[String,Int]] = Map(o1 -> Map(i1 -> 1, i2 -> 2))

scala> val m2 = Map("o1" -> Map("i2" -> 3))
m2: Map[String,Map[String,Int]] = Map(o1 -> Map(i2 -> 3))

scala> val m3 = M.op(m1, m2)
m3: Map[String,Map[String,Int]] = Map(o1 -> Map(i1 -> 1, i2 -> 5))
```

◈ 练习 10.17

为返回为monoid的函数编写monoid实例。

```
def functionMonoid[A,B](B: Monoid[B]): Monoid[A => B]
```

◈ 练习 10.18

bag的输入参数是集合，返回为Map，其中key为集合中的元素，value为元素出现的次数。如下所示：

```
scala> bag(Vector("a", "rose", "is", "a", "rose"))
res0: Map[String,Int] = Map(a -> 2, rose -> 2, is -> 1)
```

实现如下函数：

```
def bag[A](as: IndexedSeq[A]): Map[A, Int]
```

10.6.2 使用组合的 monoid 融合多个遍历

多个 monoid 可以被组合成一个，意味着折叠数据结构时可以同时执行多个计算。比如可以同时获得一个 List 的长度和总和用于计算平均值。

```
scala> val m = productMonoid(intAddition, intAddition)
m: Monoid[(Int, Int)] = $anon$1@8ff557a

scala> val p = listFoldable.foldMap(List(1,2,3,4))(a => (1, a))(m)
p: (Int, Int) = (4, 10)

scala> val mean = p._1 / p._2.toDouble
mean: Double = 2.5
```

使用 `productMonoid` 和 `foldMap` 来手动组装 `Monoid` 比较烦琐。这个问题的部分原因是需要在 `foldMap` 中构建 map 函数，然后手动保持对齐，以和 monoid 类型匹配。其实可以创建组合子库让装配这些 monoid 变得简单，定义复杂的可以在一次循环中并行计算，这样的库不在本章的讨论范围内，但可以参考本章的注释和资料。

10.7 小结

第三部分内容是让你习惯和更抽象的结构打交道，并提升能力去识别它们。本章引入了最简单的纯代数的抽象 monoid。当开始研究它时，你将发现自己的库中也可能存在大量的 monoidal 结构可以探索。可结合（associative）特质容许折叠任何可折叠的数据类型并可以灵活地进行并行处理。同样 monoid 是可组合的，可以以申明式和可重用式的方式去组合折叠。

`Monoid` 是第一个纯抽象代数，它定义为抽象的操作和相应的法则。可以在不知道参数是什么，仅仅知道其类型可以构建 monoid 的情况下编写可用的函数。在第三部分接下来的章节中将开发这种抽象的思维模式，并将考虑其他的纯代数的接口和展示使用它们封装本书中不断出现的普通模式。

11 Monad

上一章中介绍了简单的代数结构 monoid，它是第一个完全抽象、纯代数的接口，并且引导我们思考接口新方法。一个有用的接口可以被定义为一组操作和其相应的法则。

本章将继续这种思维模式，并应用它去重构第一部分、第二部分的一些库以减少代码重复。这里将发现两个新的抽象接口，`Functor` 和 `Monad`，并且获得更多的经验去发现代码中的这类抽象结构。[1]

11.1 函子：对 map 函数的泛化

在第一部分和第二部分中实现了一些不同的组合子库。在每个案例中首先编写了少许的原语（primitive），然后使用这些原语来定义组合子。这些组合子的相似性是值得注意的，比如为每个数据类型实现了 `map` 函数，用于提升某个数据类型到上下文中的数据类型。对于 `Gen`、`Parser` 和 `Option`，类型的签名如下：

```
def map[A,B](ga: Gen[A])(f: A => B): Gen[B]
def map[A,B](pa: Parser[A])(f: A => B): Parser[B]
def map[A,B](oa: Option[A])(f: A => B): Option[A]
```

这些类型签名仅仅是数据类型不同（`Gen`、`Parser` 或 `Option`）。这里将它定义为 Scala trait 并实现 `map`：

```
trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B): F[B]
}
```

这里使用类型的构造器 `F[_]` 来参数化 `map`，就像上一章中的 `Foldable`。[2] 不仅可以为 `F[_]` 挑选实际的类型，比如 `Gen` 或 `Parser`，而且 `Functor` trait 是参数化的，以下是 `List` 的一个实例：

---

1 名称 functor 和 monad 源自于数学分支 category 理论，但不需要任何 category 理论的背景去了解本章的内容。如果感兴趣可以在本章的笔记中跟踪参考链接发现更多信息。

2 类型构造器是作用在类型上并构造一个类型。比如，`List` 是一个类型构造器，不是类型。没有类型 `List` 对应的值，但是可以使用 `Int` 构造类型 `List[Int]`。同样，`Parser` 也可以传入 `String` 来构造 `Parser[String]`。

```
val listFunctor = new Functor[List] {
    def map[A,B](as: List[A])(f: A => B): List[B] = as map f
}
```

这里类型构造器比如 List（或 Option，或 F）是一个 functor，而且 Functor[F] 实例的构成也证明 F 的确是一个 functor。

这个抽象有什么作用？就像在书中多次提及的那样，使用接口定义的操作并以一个纯代数的方式探索可用的函数。这里稍作停留来探索使用 map 可以定义哪些可用的操作。

从一个例子开始，假如有 F[(A, B)]，F 是一个 functor，我们可以“分派”（distribute）F 到 pair，从而生成 (F[A], F[B])：

```
trait Functor[F[_]] {
    ...
    def distribute[A,B](fab: F[(A, B)]): (F[A], F[B]) =
        (map(fab)(_._1), map(fab)(_._2))
}
```

这样实现仅仅是从抽象类型出发，现在来考虑一下对实际类型的意义，比如说 List、Gen、Option，等等。比如，假如分派一个 List[(A, B)]，将得到两个相同长度的 List，一个包含所有类型 A 的元素，而另外一个包含所有的 B 元素。这个操作实际上被称为 *unzip*。所以这里实现了一个泛化的 unzip 函数，它不仅适用于 List，而且适用于所有的 functor。

实现这样的操作的同时，应该探索一下是否可以通过 sum 或 coproduct 构建相反的操作：

```
def codistribute[A,B](e: Either[F[A], F[B]]): F[Either[A, B]] = e match {
    case Left(fa) => map(fa)(Left(_))
    case Right(fb) => map(fb)(Right(_))
}
```

对 Gen 来说 codistribute 的意义何在？假如有一个对 A 的生成器或对 B 的生成器，就可以构建一个生成 A 或 B 的生成器，具体生成哪个取决于拥有哪一个生成器。

至此实现了两个实际可用的 combinator，它们是基于纯的 Functor 抽象接口实现的，而且可以在各种实现了 map 的数据类型中使用。

11.1.1 函子法则

无论何时创建一个类似 Functor 的抽象，都不仅需要考虑需要实现哪些抽象方法，而且需要考虑遵循哪些法则。遵循什么样的法则完全由你决定，[3]Scala 不会强加任何这样的法则。这里法则之所以重要是因为：

- 法则帮助接口定义了语法，其代数定义可以在实例间被独立地推演。比如，当使用 Monoid[A] 和 Monoid[B] 去构建一个 Monoid[(A,B)] 时，monoid 法则保证的结论是，“融合”后的 monoid 操作仍然是可结合的（associative）。这不需要知道有关 A 和 B 的任何其他细节。

3 因为借用了已经存在的数据抽象名称，如 functor 或 monoid，所以建议使用的法则应该也是数学定义的。

- 更具体一点，经常需要依赖法则基于类似 Functor 这样抽象接口中的函数去编写各种组合子。这将在随后的例子中展示。

对于 Functor，将采用类似在第 7 章为 Par 数据类型引入的法则：[4]

```
map(x)(a => a) == x
```

换言之，使用 identity 函数映射一个结构 x 结果应该仍然是 x。这个法则十分自然，并且都能满足其他类型。这个法则（parametricity 的结果）定义了 map(x) 保持 x 的结构不变，对具体的实现提出了一些限制，不可以抛出异常，移除 list 的第一个元素，将 Some 转化为 None，等等。仅仅结构中的元素可以被 map 修改，结构本身和形状保持不变。这个法则普遍存在于 List、Option、Par、Gen 和定义 map 的其他数据类型中。

给出一个具体的保持结构不变的例子，可以考虑之前定义的 distribute 和 codistribute：

```
def distribute[A,B](fab: F[(A, B)]): (F[A], F[B])

def codistribute[A,B](e: Either[F[A], F[B]]): F[Either[A, B]]
```

因为只知道 F 是一个 functor，法则确保返回值和参数保持一样的形状。假如 distribute 的输入是一组数据对（pair），那么返回的两个列表将和输入列表长度相同，同时相应的元素将以同样的顺序出现。这样的代数推导潜在地帮我们节省了很多工作，因为不再需要单独测试这些特性。

11.2 Monad：对 flatMap 和 unit 函数的泛化

Functor 只是重构的库中众多抽象中的一个。Functor 的作用不是那么显著，是因为仅仅使用纯的 map 定义不出太多可用的操作。接下来将研究一个有趣的接口，Monad。使用这个接口可以实现很多可用的操作并且一劳永逸地对重复的代码进行重构。同时相关的法则可以推导库按照期望运行。

回想迄今本书中提及的一些数据结构，map2 是可以同时提升（lift）两个参数的函数。对于 Gen、Parser 和 Option 函数，map2 可以按如下实现：

示例 11.1 为 Gen、Parser 和 Option 实现 map2

```
def map2[A,B,C](                    <——通过调用函数f组合随机生成器fa和fb的运行结果，生成了类型C的随机生成器。
        fa: Gen[A], fb: Gen[B])(f: (A,B) => C): Gen[C] =
    fa flatMap (a => fb map (b => f(a,b)))
def map2[A,B,C](                    <——通过调用函数f组合解析器fa和fb的运行结果，生成了类型C的解析器。
        fa: Parser[A], fb: Parser[B])(f: (A,B) => C): Parser[C] =
    fa flatMap (a => fb map (b => f(a,b)))
def map2[A,B,C](                    <——假如两个Option都有值，使用函数f组合两个Options，否则返回None。
        fa: Option[A], fb: Option[B])(f: (A,B) => C): Option[C] =
    fa flatMap (a => fb map (b => f(a,b)))
```

4 这个法则也是源自于数学定义的 functor。

这些函数不仅仅名字是相同的。尽管被操作的数据类型看起来没有任何关系，但是实现是相同的！唯一不同的是被操作的数据类型不同。这也证实了我们的猜想，它们是一个更抽象的模式的实例，应该进行相关的开发以避免自我重复。比如，应该能够编写一个 map2，然后被所有的这些类型重用。

这里通过相同的函数名和同样顺序的参数让重复代码变得十分明显。在每天的实际工作中这可能比较困难。但是随着更多的库的编写你将可以更好地辨别这些模式并重构成普通的抽象。

Monad 特质

这里把 Parser、Gen、Par、Option 和其他数据类型统一起来的是 monad。就像定义的 Functor 和 Foldable，可以统一为 Monad 定义一个 Scala trait，其中包括 map2 和一些其他的函数，而不是为每个具体类型重复定义。从本书第二部分关心每个单独的数据类型中寻找并定义最小的一个原始的操作集合，在这之上通过组合定义更多可用的操作。这里将采用同样的方法来改善抽象接口以获取一个原始操作的集合。

现在引入一个名叫 Mon 的新 trait，因为最终需要定义 map2，先把它加入。

示例 11.2 为 map2 创建一个 Mon trait

```
trait Mon[F[_]] {
  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A,B) => C): F[C] =
    fa flatMap (a => fb map (b => f(a,b)))
}
```

这不能编译，因为map和flatMap在上下文中没有定义。

这里仅仅实现了 map2，把 Parser、Gen 和 Option 变成一个多态的 F，[5] 但在这个多态上下文中是编译不了的。在不知道 F 任何细节的情况下，就不知道如何对 F[A] 进行 flatMap 或 Map。

最简单的操作是将 flatMap 和 Map 添加到 Mon 接口中，然后保持抽象。调用这些函数的语法变化了一点（我们不能再使用 infix 语法），但是结构还是一样的。

示例 11.3 添加 map 和 flatMap 到我们的 trait 中

```
trait Mon[F[_]] {
  def map[A,B](fa: F[A])(f: A => B): F[B]
  def flatMap[A,B](fa: F[A])(f: A => F[B]): F[B]

  def map2[A,B,C](
    fa: F[A], fb: F[B])(f: (A,B) => C): F[C] =
      flatMap(fa)(a => map(fb)(b => f(a,b)))
}
```

在接口Mon中调用抽象的函数map和flatMap。

这样的转化相当机械。仅仅检查 map2 的实现，然后将所调用的函数 map 和 flatMap 作为抽象的方法都添加到接口中。现在可以编译了，但在宣称胜利之前先定义实例

5 称类型参数为 F 是任意的行为。虽然通过约定可以称这个参数为 Foo、w00t 或 Blah2，但通常给予类型参数一个大写字母的名字，比如说 F、G 和 H，或者某些时候 M 和 N，或 P 和 Q。

Mon[List]、Mon[Parser]、Mon[Option]，等等。现在检查是否可以改进原始的操作集合。当前的原始操作集合是 map 和 flatMap，从它们可以衍生出 map2。那么 map 和 flatMap 是不是最小的原始操作集合？那些实现 map2 的数据类型都有一个 unit，并且 map 可以用 flatMap 和 unit 实现。比如，对 Gen：

```
def map[A,B](f: A => B): Gen[B] = flatMap(a => unit(f(a)))
```

那么现在选择 unit 和 flatMap 作为最小的集合。将由这些函数定义的所有数据类型都统一在一个概念下。这个 trait 称为 Monad，它包括 flatMap 和 unit 的抽象，并且提供默认的 map 和 map2 的实现。

示例 11.4 创建 Monad trait

```
trait Monad[F[_]] extends Functor[F] {

  def unit[A](a: => A): F[A]

  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]

  def map[A,B](ma: F[A])(f: A => B): F[B] =
    flatMap(ma)(a => unit(f(a)))
  def map2[A,B,C](ma: F[A], mb: F[B])(f: (A, B) => C): F[C] =
    flatMap(ma)(a => map(mb)(b => f(a, b)))
}
```

因为Monad提供了缺省的map实现，它可以继承Functor。所有的monad是functor，但不是所有的functor是monad。

monad 的名字

可以给 monad 起任何名字，比如 FlatMappable、Unicorn 或 Bicycle。但是 monad 已经是完美的名字了。它来自于 category 理论，一些数学概念引发了很多函数式编程的观点。之所以叫 monad 是有意和 monoid 类似的，它们在比较深层次的地方是有关联的。可以查看本章的笔记获取更多的内容。

将具体的数据类型联系起来为 Gen 实现 Monad 接口。

示例 11.5 对 Gen 实现 Monad

```
object Monad {
  val genMonad = new Monad[Gen] {
    def unit[A](a: => A): Gen[A] = Gen.unit(a)
    def flatMap[A,B](ma: Gen[A])(f: A => Gen[B]): Gen[B] =
      ma flatMap f
  }
}
```

这里只需要实现 unit 和 flatMap，然后不需要任何代价得到 map 和 map2。这样一劳永逸地为任何可以提供 Monad 实例的数据类型实现了它们！这仅仅是刚刚开始，将会有更多函数可以一劳永逸地实现。

◈ 练习 11.1

为 Par、Parser、Option、Stream 和 List 编写 monad 实例。

◇ 练习 11.2

难：State 看起来也是 monad，但需要接收两个类型参数。这就需要一个参数的类型构造器来实现 Monad。尝试实现 State monad，你将会遇到一些问题，考虑可能的解决方法。下一章中将会讨论具体的解决方法。

11.3 Monadic 组合子

现在已经有了 monad 的原始语义，回看之前章节查看是否有其他为 monadic 数据类型实现的函数可以统一被实现。

◈ 练习 11.3

大家已经很熟悉 sequence 和 traverse 组合了，之前的章节有很多地方都实现了它，现在使用 Monad[F] 统一实现它们。

```
def sequence[A](lma: List[F[A]]): F[List[A]]
def traverse[A,B](la: List[A])(f: A => F[B]): F[List[B]]
```

之前的一个组合是为 Gen 和 Parser 实现 listofN，它复制 parser 或 generator n 次，然后得到一个同样长度的 List 的 parser 或 generator。可以为所有的 F 实现这个组合，只要把它加入 Monad trait 中。这里应该给它一个更加抽象的名字，比如 replicateM（意思是在 monad 中复制）。

◈ 练习 11.4

实现 replicateM。

```
def replicateM[A](n: Int, ma: F[A]): F[List[A]]
```

◈ 练习 11.5

考虑 replicateM 对不同数据类型的表现行为。比如，对 List monad 的表现如何？那么 Option 呢？使用自己的语言描述 replicateM 的一般含义。

同样有一个对 Gen 数据类型的组合 product，它接收两个 generator，然后转化成一个 pair 的 generator，之前对 Par 也有同样的操作。这两种情况下都是用 map2 实现的，因此也可以为任何 monad F 实现它：

```
def product[A,B](ma: F[A], mb: F[B]): F[(A, B)] = map2(ma, mb)((_, _))
```

不必限制在所见的组合中，重要的是用于尝试并发现新的组合。

◈ 练习 11.6

难：这是一个之前没有遇到过的函数。实现函数 filterM。它看起来和 filter 类似，只是接收的不是函数 A=>Boolean，而是 A=>F[Boolean]（将普通的这样的函数取代为 monadic 等式常常会产生有趣的结果）。实现这个方法并考虑其对不同数据类型意味着什么。

```
def filterM[A](ms: List[A])(f: A => F[Boolean]): F[List[A]]
```

这里的组合只是 Monad 可以实现的一小部分，我们将在第 13 章中看到更多。

11.4 单子定律

这节中将引入法则去管理 Monad 接口。[6] 毫无疑问也期望 functor 法则对 Monad 是成立的，因为 Monad[F] 是一个 Functor[F]，但是除此之外呢？什么样的法则可以约束 flatMap 和 unit？

11.4.1 结合法则

举个例子，假如需要结合三个 monadic 值并生成一个，哪两个先结合呢？这个有关系吗？要回答这个问题，先抽象地查看一个具体的 Gen monad 例子。

假如测试一个产品订单系统，需要 mock 一些订单。可能有一个 Order case class 和此类型的 generator。

示例 11.6 定义 Order class

```
case class Order(item: Item, quantity: Int)
case class Item(name: String, price: Double)

val genOrder: Gen[Order] = for {
    name <- Gen.stringN(3)      ←——长度为3的随机串。
    price <- Gen.uniform.map(_ * 10)   ←——0和10之间的均匀分布的随机双精度浮点数。
    quantity <- Gen.choose(1,100)   ←——0和100之间的随机整数。
} yield Order(Item(name, price), quantity)
```

这里内联地生成 Item（从 name 到 price），但是可能有其他地方需要单独生成 Item，所以将它引入到自己的 generator：

```
val genItem: Gen[Item] = for {
    name <- Gen.stringN(3)
    price <- Gen.uniform.map(_ * 10)
} yield Item(name, price)
```

6 再次，这些法则源自 category 理论里的 monads 观点，但 category 理论的背景对理解这一节不是必需的。

这样可以在 genOrder 中使用它：

```
val genOrder: Gen[Order] = for {
    item <- genItem
    quantity <- Gen.choose(1,100)
} yield Order(item, quantity)
```

此时它们应该做一样的事情，对吗？看起来可以这么假设。但是不要急，怎么能够确定，毕竟它们不是一模一样的代码。

将两种实现都展开为对 map 和 flatMap 的调用，以帮助查看怎么回事？前一个方法，转化是很直观的：

```
Gen.nextString.flatMap(name =>
Gen.nextDouble.flatMap(price =>
Gen.nextInt.map(quantity =>
    Order(Item(name, price), quantity))))
```

第二种方法看起来如下（内联调用 genItem）：

```
Gen.nextString.flatMap(name =>
Gen.nextInt.map(price =>
    Item(name, price))).flatMap(item =>
Gen.nextInt.map(quantity =>
    Order(item, quantity)
```

一旦展开后很清楚地发现两个实现过程不是相同的。然而 for 推导时，又是假设两个实现做了一样的事情，结果是一致的。事实上，如果它们结果不一致才令人感到惊讶和奇怪，因为我们默认了 flatMap 遵循结合法则：

```
x.flatMap(f).flatMap(g) == x.flatMap(a => f(a).flatMap(g))
```

这个法则对所有的 x、f 和 g 均成立，而且类型不仅是 Gen，对 Parser、Option 和其他 monad 也同样成立。

11.4.2 为指定的 monad 证明结合法则

这里先证明这个法则对 Option 是成立的。所要做的就是将先前等式中的 x 替换成 None 或者 Some(v)。

从 x 是 None 开始，那么等号两边都替换成 None：

```
None.flatMap(f).flatMap(g) == None.flatMap(a => f(a).flatMap(g))
```

因为对所有的 f None.flatMap(f) 都是 None，所以简化为：

```
None == None
```

因此，法则在 x 是 None 的时候成立。那么当 x 是 Some(v) 时（其中 v 是任意的选择）的情况是：

```
x.flatMap(f).flatMap(g)        == x.flatMap(a => f(a).flatMap(g))   <——原始的法则。
Some(v).flatMap(f).flatMap(g)  == Some(v).flatMap(a => f(a).flatMap(g))
                                                  两边同时使用Some(v)替代x。
f(v).flatMap(g)                == (a => f(a).flatMap(g))(v)
                                                  使用Some(v).flatMap(...)的定义。
f(v).flatMap(g)                == f(v).flatMap(g)   <——简化函数(a => ..)(v)。
```

因此，法则同样在 x 是任意 Some(v) 时成立。证明完毕，因为法则对 x 是 None 和 Some 时都成立，并且 Option 只有这两种可能。

KLEISLI 组合：结合律更清晰的视图

刚才讨论的并不容易看出是在说结合律法则，记得之前 monoid 的结合律法则吗？它是很清晰的法则：

```
op(op(x,y), z) == op(x, op(y,z))
```

但是 monad 的结合法则看起来不像这个这么清晰！幸运的是，有一种方式可以让它更清晰。不考虑 F[A] 类型的 monadic 值，而是考虑 A => F[B] 的 monadic 函数。这样的函数称之为 Kleisli 箭头。[7] 它们是可以互相组合的：

```
def compose[A,B,C](f: A => F[B], g: B => F[C]): A => F[C]
```

◈ **练习 11.7**

实现 Kleisli composition 函数 compose。

现在可以以一个更加对称的方式为 monad 声明结合法则了：

```
compose(compose(f, g), h) == compose(f, compose(g, h))
```

◈ **练习 11.8**

难：使用 compose 实现 flatMap，这看起来是另一组 monad 组合子的最小集合：compose 和 unit。

◇ **练习 11.9**

展示使用 flatMap 和 compose 构建 associative 法则是等同的。

11.4.3 单位元法则

其他的 monad 法则就很容易查看了。就像 zero 是 monoid 的 append 方法的一个*单位元*（identity）*元素*，monad 的 compose 函数也有一个单位元元素。事实上，这正是 unit[8] 函数，并且这也是叫这个名字的原因：[9]

```
def unit[A](a: => A): F[A]
```

这个函数返回的类型作为正确的参数传递给 compose。[10] 任何对象和 unit 组合返回都是同样的对象。通常这可以表达为两种法则，*左 identity* 和*右 identity*：

```
compose(f, unit) == f
compose(unit, f) == f
```

---

7 Kleisli 箭头源自 category 理论，并以瑞士数学家 Heinrich Kleisli 命名。

8 unit 是单元的意思，不要跟 Unit 类型混淆。——译者注

9 unit 的名字经常被用在数学中去表示某些操作单元。

10 Scala 中非严格 A to F[A]((=> A)=>F[A])和普通的 A=>F[A] 是有区别的，虽然这里不做区分。

同样可以用 flatMap 表述这个法则，不过就没有那么清晰了：

```
flatMap(x)(unit) == x
flatMap(unit(y))(f) == f(y)
```

◈ **练习 11.10**

证明 identity 法则的两个声明是等同的。

◈ **练习 11.11**

证明你所选择的 monad 实例遵循 identity 法则。

◈ **练习 11.12**

第三种 monadic 组合的最小集合 map、unit 和 join。使用 flatMap 实现 join。

```
def join[A](mma: F[F[A]]): F[A]
```

◈ **练习 11.13**

使用 join 和 map 实现 flatMap 或 compose。

◇ **练习 11.14**

使用 join、map 和 unit 重新陈述 monad 法则。

◇ **练习 11.15**

使用自己的语言解释对于 Par 和 Parser associative 来说法则意味着什么。

◇ **练习 11.16**

使用自己的语言解释对于 Gen 和 List 来说 identity 法则意味着什么。

11.5 什么是 monad

从更广阔的视角来查看 Monad 接口不寻常的地方。monad 实例化的数据类型看起来没有什么关联，Monad 从它们中抽出了重复的代码，但是准确地说什么是 monad？ monad 意味着什么？

你可能认为接口是为一个抽象的数据类型提供一组相对完备的 API，仅仅是对指定类型的抽象。毕竟一个单向链表和一个数组链表可能在幕后的实现是不同的，但是又分享同样的接口，所以在编写具体的应用程序时是非常方便的。Monad 和 Monoid 一样是一个更抽象、纯代数的接口。Monad 组合通常是一个 monad 数据类型所有 API 中的一小部分，Monad 不是对一个类型的泛化，而是大量不同的数据类型满足 Monad 接口和法则的抽象。

已经看到三组最小的原始 Monad 组合，Monad 实例需要提供实现下面的一组：

- unit and flatMap
- unit and compose
- unit, map, and join

并且两个 monad 法则需要被满足，associativity 和 identity，它们可以以不同的方式公式化。因此我们可以简单地描述 monad：

monad 是一个满足 associativity 和 identity 法则的 monadic 组合的最小集的实现。

这是十分体面、精确和简短的定义。精确地说这也是唯一正确的定义。monad 被它的操作和法则精确地定义，不多不少。但是并不满足于此，因为这个定义没有包含 monad 意味着什么。原因是这是一个自包含的定义。如果是一个编程的新手，这个定义与你所获得的编程知识没有半点的关联。

为了真正理解 monad 是怎么回事，把 monad 与已经熟知的东西联系到一起，把它和一个更广阔的上下文联系在一起。接下来看一些 monad 实例并比较它们，以获得一些直观上的理解。

11.5.1 identity monad

为了提炼 monad 的本质，先查看最简单有趣的例子，identity monad。

```
case class Id[A](value: A)
```

◈ **练习 11.17**

为这个类型实现 map 和 flatMap 方法，并且实现 Monad[Id]。

现在，Id 仅仅是一个简单的包装。它没有添加任何别的东西。将 Id 应用在 A 上是完全相等的，因为包装和解包是完全等同的（我们可以在不丢失任何信息的情况下从一个类型到另一个类型）。但是 identity *monad* 的意义何在？让我们在 REPL 中使用它：

```
scala> Id("Hello, ") flatMap (a =>
     |   Id("monad!") flatMap (b =>
     |     Id(a + b)))
res0: Id[java.lang.String] = Id(Hello, monad!)
```

用 for 推导来表达会更清晰：

```
scala> for {
     |   a <- Id("Hello, ")
     |   b <- Id("monad!")
     | } yield a + b
res1: Id[java.lang.String] = Id(Hello, monad!)
```

那么 identity monad 的 flatMap 具体做了什么？它只是简单的变量替换。变量 a 和 b 被各自绑定到 "Hello, " 和 "monad!"，然后分别替换到表达式 a+b。可以不用 Id 包装做相同的事情。如下所示仅仅使用 Scala 自己的变量：

```
scala> val a = "Hello, "
a: java.lang.String = "Hello, "

scala> val b = "monad!"
b: java.lang.String = monad!

scala> a + b
res2: java.lang.String = Hello, monad!
```

除了 Id 类型的包装，没有任何不同。那么现在至少可以对什么是 monad 得到部分答案。可以说 monad 提供了一个引入和绑定变量的上下文，同时执行了变量替换。

接下来看看能否得到剩余答案。

11.5.2 状态 monad 和 partial type application

回顾在第 6 章中讨论的 State 数据类型，那时为 State 实现的一些组合，包括 map 和 flatMap。

示例 11.7 回顾状态与数据类型

```
case class State[S, A](run: S => (A, S)) {
  def map[B](f: A => B): State[S, B] =
    State(s => {
      val (a, s1) = run(s)
      (f(a), s1)
    })
  def flatMap[B](f: A => State[S, B]): State[S, B] =
    State(s => {
      val (a, s1) = run(s)
      f(a).run(s1)
    })
}
```

看起来 State 十分适合实现 monad。但是它的类型构造器需要两个类型参数，Monad 只需要一个。那么不能简单地说 Monad[State]。但是如果选定一个具体的 S，那么就会有 State[S, _]，这是 Monad 所期望的。所以 State 不仅仅是一个 monad 实例，而是整个家族，对每一个 S 都是一个。可以为 State 部分地将 S 固定为一些具体的类型，就像部分应用函数。

这就像函数的部分应用，除了是在类型上。比如可以创建一个 IntState 类型构造子，只要将 State 的第一个类型参数固定为 Int：

```
type IntState[A] = State[Int, A]
```

IntState 正是构建 Monad 所需的：

```
object IntStateMonad extends Monad[IntState] {
  def unit[A](a: => A): IntState[A] = State(s => (a, s))
  def flatMap[A,B](st: IntState[A])(f: A => IntState[B]): IntState[B] =
    st flatMap f
}
```

当然，假如需要为每个特定的 state 类型手动地编写一个 Monad 实例，那会变得过于重复。不幸的是，Scala 不允许像创建匿名函数那样使用下画线语法 State[Int, _] 去创建匿名的类型构造子。但是可以在类型的层次使用类似 lambda 语法。比如，可以直接内联声明 IntState：

```
object IntStateMonad extends
    Monad[({type IntState[A] = State[Int, A]})#IntState] {
    ...
}
```

第一次看到这个语法时会觉得不那么舒服。但是这里仅仅是在括号里声明了匿名类型。匿名类型有一个类型别名的成员 IntState，和之前是一样的。在括号外使用语法 # 去访问它的 IntState 成员。就像使用点（.）去访问一个值的成员，可以使用符号 # 去访问一个类型的成员（参照 Scala 语言规范 Type Projection 部分〔http://mng.bz/u70U〕）。

一个类型构造器像这样内联的申明在 Scala 中称之为一个类型 *lambda*。可以使用这个小技巧去部分应用 State 类型构造器，并且申明一个 StateMonad trait。一个 StateMonad[S] 实例是一个给定类型 S 的 monad 实例。

```
def stateMonad[S] = new Monad[({type f[x] = State[S,x]})#f] {  ←——f的名字选择是任意的。
    def unit[A](a: => A): State[S,A] = State(s => (a, s))
    def flatMap[A,B](st: State[S,A])(f: A => State[S,B]): State[S,B] =
        st flatMap f }
```

再次，只需要给出 unit 和 flatMap 的实现，其他 monadic 组合都可以容易地实现。

◈ 练习 11.18

现在有了 State monad，你应该尝试理解其行为是什么。State monad 中的 replicateM 含义是什么？ map2？ sequence？

现在查看一下 Id monad 和 State monad 的区别。State 中（除了 unit 和 flatMap）其他原始的操作还有用于读取当前状态的 getState 和设置新状态的 setState：

```
def getState[S]: State[S, S]
def setState[S](s: => S): State[S, Unit]
```

这些 combinator 构建了 State 的最小原始操作，它们和 monadic 原始操作（unit 和 flatMap）一起构成了 State 数据类型的所有操作。monad 一般都是这样的，它们都包括 unit 和 flatMap，并且每个 monad 又有自己额外的原始操作。

◈ 练习 11.19

getState、setState、unit 和 flatMap 共同保持了什么法则？

State *monad* 的意义是什么？现在来看一个简单的例子，代码的细节不重要，要注意在 for 代码块中使用 getState 和 setState。

示例 11.8 在 for 推导中获取和设置状态

```
val F = stateMonad[Int]

def zipWithIndex[A](as: List[A]): List[(Int,A)] =
    as.foldLeft(F.unit(List[(Int, A)]()))((acc,a) => for {
        xs <- acc
        n  <- getState
        _  <- setState(n + 1)
    } yield (n, a) :: xs).run(0)._1.reverse
```

这个函数使用 State 动作对 list 中所有的元素进行计数。State 是步数的累加值。我们从 0 开始运行整个 State 动作，因为是以相反的顺序执行的，所以最后将结果反转。[11]

注意在 for 推导中 getState 和 setState 到底做了些什么？明显的事情是像 Id monad 那样进行变量绑定，按顺序将每个 State（getState、acc、setState）和变量绑定。但是代码中发生了更多的事情，在 for 推导的每一行，flatMap 的实现保证了当前的 State 是可以被 getState 调用的，同时新的 State 可以在 setState 中被赋值并进行随后的动作。

Id monad 和 State monad 的不同行为展示了 monad 的一般特质。这里看到 flatMap 的调用链（或者是 for 推导）像一个命令式程序，程序的语句是变量赋值，*monad 指定了这些语句的边界*。比如，对于 Id，仅仅是在 Id 的构造子中进行包装和解包。对于 State，最新的 State 从一个语句传递到另一个。对于 Option monad，一个语句可能返回 None 并终止程序。对于 List monad，一个语句可能返回多个结果，这会引起随后的代码可能执行多次，每次对应一个结果。

Monad 合约没有指定代码直接发生什么，仅仅指定发生的事情需要满足 associativity 和 identity 法则。

◈ 练习 11.20

难：为了巩固对 monad 的理解，对如下的类型实现 monad 实例并解释它的含义。它包括哪些原始的操作？ flatMap 的动作是什么？ sequence、join 和 replicateM 的含义是什么？它如何实现 monad 法则？ [12]

```
case class Reader[R, A](run: R => A)

object Reader {
    def readerMonad[R] = new Monad[({type f[x] = Reader[R,x]})#f] {
        def unit[A](a: => A): Reader[R,A]
        def flatMap[A,B](st: Reader[R,A])(f: A => Reader[R,B]): Reader[R,B]
    }
}
```

11 这个比在循环中追加链表尾部的做法要快一些。

12 在本章笔记中查看此数据类型的更多讨论。

11.6 小结

本章将本书中出现的模式统一到一个概念下：monad。这允许我们一劳永逸地对很多看起来没有共性的数据类型编写多个组合子。这里从不同的角度讨论了它们所满足的法则，monad 法则，并且洞察它的含义。

像这样抽象的话题是不可能一下子彻底明白的。它需要你从不同方面不断地审视，逐步认识。当你发现新的 monad，或者 monad 的新应用，抑或在新的上下文中发现它时，你都会对它获得新的认识。每一次你都可能会这样想：“我本以为已经理解 monad，但现在才真正理解”。

12 可应用和可遍历函子

在上一章中为不同的组合子库编写的很多函数可以表达为单一的接口 Monad。Monad 提供了一个强大的接口，事实证明可以使用 flatMap 以纯函数式方法编写命令式程序。

本章将要学习相关的抽象，*可应用函子*，虽没有 Monad 那么强大，但更普遍（因此更通用）。在寻找可应用函子的过程中，也展示了*如何发现这种抽象*并利用这种方式发现另外一种有用的抽象，*可遍历函子*。这些抽象需要一些时间去融会贯通，但稍加注意的话，将会发现在日常函数式编程中它们不断出现。

12.1 泛化单子

至此已经看到了不同的操作，比如 `sequence` 和 `traverse`，它们在不同的 monad 中被实现了多次。上一章中泛化了这个实现，让它们对任何 monad `F` 都有效：

```
def sequence[A](lfa: List[F[A]]): F[List[A]]
    traverse(lfa)(fa => fa)

def traverse[A,B](as: List[A])(f: A => F[B]): F[List[B]]
    as.foldRight(unit(List[B]()))((a, mbs) => map2(f(a), mbs)(_ :: _))
```

这里使用 `map2` 和 `unit` 实现了 `traverse`，`map2` 也可以使用 `flatMap` 实现：

```
def map2[A,B,C](ma: F[A], mb: F[B])(f: (A,B) => C): F[C] =
    flatMap(ma)(a => map(mb)(b => f(a,b)))
```

可能没有注意的是很多 monad 组合可以使用 `unit` 和 `map2` 来定义。组合 `traverse` 是一个例子，它没有直接调用 `flatMap`，以至于我们应该考虑 `map2` 是原语（primitive）还是衍生的（derived）。更进一步说，对一些数据类型 `map2` 可以不使用 `flatMap` 来实现。

所有的这些都表明 `Monad` 的变化，`Monad` 使用 `unit` 和 `flatMap` 作为原语（primitive），然后衍生出 `map2`。但是可以获得*另外一个*抽象，采用 `unit` 和 `map2` 作为原语。将看到的这个抽象称为*可应用函子*，虽然没有 monad 那么强大，但也会带来一些额外功能。

12.2 Applicative trait

Applicative 函子可以被捕捉为一个接口，Applicative 中的原语包括 map2 和 unit。

示例 12.1 创建 Applicative 接口

```
trait Applicative[F[_]] extends Functor[F] {
  // primitive combinators
  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C]
  def unit[A](a: => A): F[A]

  // derived combinators
  def map[B](fa: F[A])(f: A => B): F[B] = ←——可以使用unit和map2实现map。
    map2(fa, unit(()))((a, _) => f(a)) ←——()是类型Unit的唯一值，unit(())是使用无用值()调用unit。

  def traverse[A,B](as: List[A])(f: A => F[B]): F[List[B]] ←traverse的定义是相同的。
    as.foldRight(unit(List[B]()))((a, fbs) => map2(f(a), fbs)(_ :: _))
}
```

所有的 *Applicative* 都是 *functor*。这里用 map2 和 unit 实现了 map，就像之前对特定的数据类型那样。这里的实现对将要查看的 Applicative 法则有一定的指导性，因为像 Functor 法则一样 map 的实现保持数据结构不变。

注意 traverse 的实现没有变化，同样可以将其他不依赖于 flatMap 和 join 的函数移入 Applicative 中。

◈ 练习 12.1

尽可能多地将 Monad 组合子移到 Applicative 组合子，只使用 map2 和 unit 函数，或根据这两个实现的其他方法。

```
def sequence[A](fas: List[F[A]]): F[List[A]]
def replicateM[A](n: Int, fa: F[A]): F[List[A]]
def product[A,B](fa: F[A], fb: F[A]): F[(A,B)]
```

◈ 练习 12.2

难：可应用（applicative）这个名词源于这样一个事实，它使用另一组原语 unit 和 apply 函数来表达 Applicative 接口，[1] 而非 unit 和 map2 函数。展示一下这种表达方式跟按照 unit 和 apply 所定义的 map2 和 map 在表达上是等价的。同时确认 apply 函数是可以根据 map2 和 unit 实现的。

```
trait Applicative[F[_]] extends Functor[F] {

  def apply[A,B](fab: F[A => B])(fa: F[A]): F[B] ←——使用map2和unit定义。
  def unit[A](a: => A): F[A]
```

1 applicative 表示接口包含 apply 函数。——译者注

```
  def map[A,B](fa: F[A])(f: A => B): F[B]
  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A,B) => C): F[A]
}
```

使用apply和unit定义。

◈ 练习 12.3

apply 方法对实现 map3、map4 等方法也很有用，模式很简单。实现 map3、map4 只需要使用 unit、apply 以及可用于函数的柯里化方法。[2]

```
def map3[A,B,C,D](fa: F[A],
                  fb: F[B],
                  fc: F[C])(f: (A, B, C) => D): F[D]

def map4[A,B,C,D,E](fa: F[A],
                    fb: F[B],
                    fc: F[C],
                    fd: F[D])(f: (A, B, C, D) => E): F[E]
```

更进一步，可以让 Monad[F] 成为 Applicative[F] 的一个子类，并使用 flatMap 实现 map2。这样所有的 *monad* 都是可应用函子，所以对已经是 monad 的数据类型我们不需要分别提供 Applicative 实例。

示例 12.2 让 Monad 成为 Applicative 的子类型

```
trait Monad[F[_]] extends Applicative[F] {
  def flatMap[A,B](fa: F[A])(f: A => F[B]): F[B] = join(map(fa)(f))

  def join[A](ffa: F[F[A]]): F[A] = flatMap(ffa)(fa => fa)

  def compose[A,B,C](f: A => F[B], g: B => F[C]): A => F[C] =
    a => flatMap(f(a))(g)

  def map[B](fa: F[A])(f: A => B): F[B] =
    flatMap(fa)((a: A) => unit(f(a)))

  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C] =
    flatMap(fa)(a => map(fb)(b => f(a,b)))
}
```

Monad的最小实现须包含unit，重载flatmap或join和map。

到目前只是按照类型的签名重新编排 API 的函数，现在来理解 Monad 和 Applicative 的不同之处。

2 假设 f:(A,B)=>C，f.curried 的类型是 A=>B=>C。在 Scala 中 curried 的方法存在于任何元的函数中。

12.3 单子与可应用函子的区别

上一章中一个 Monad 可以定义为不同的最小的操作集合：

- unit 和 flatMap
- unit 和 compose
- unit、map 和 join

那么是否 Applicative 操作 unit 和 map2 也是一组 monad 的最小操作集合呢？答案是否定的，例如像 join 和 flatMap 这样的 monadic 组合子就不能只用 map2 和 unit 实现。下面以 join 为例：

```
def join[A](f: F[F[A]]): F[A]
```

仅仅依据代数的推导，就可以发现 unit 和 map2 不可能实现这个函数。join 函数“去除了一层” F。但是 unit 函数仅仅只能加一层 F，map2 只是在 F 中应用一个函数操作而不去做 flatten 这样减少层数的操作。使用同样的论证，可以发现 Applicative 同样没有办法实现 flatMap。

因此，Monad 显然是在 Applicative 之外添加了额外的功能。这些是什么？让我们查看一个具体的实例。

12.3.1 对比 Option applicative 与 Option monad

假如使用 Option 去作用于两个 Map object 的查找结果，简单地组合两个独立的查找结果 map2 就足够了。

示例 12.3 使用 Option applicative 组合结果

```
val F: Applicative[Option] = ...

val depts: Map[String,String] = ...          <——部门，按员工名索引。
val salaries: Map[String,Double] = ...       <——薪水，按员工名索引。
val o: Option[String] =
    F.map2(depts.get("Alice"), salaries.get("Alice"))(
        (dept, salary) => s"Alice in $dept makes $salary per year"   <——
    )                                                  字符串中的变量dept和salary会被替换。
```

这里两个查找是独立的，仅仅是在 Option 中组合了它们的结果。假如需要一个查找的结果影响随后的查找结果，那么需要 flatMap 或 join，如下面所描述的一样。

示例 12.4 使用 Option monad 组合结果

```
val idsByName: Map[String,Int]  <——员工ID，按员工名索引。
val depts: Map[Int,String] = ...  <——部门，按员工ID索引。
val salaries: Map[Int,Double] = ...  <——薪水，按员工ID索引。
val o: Option[String] =
    idsByName.get("Bob").flatMap { id =>  <——查询Bob的ID，然后使用结果去进一步查询。
        F.map2(depts.get(id), salaries.get(id))(
```

```
      (dept, salary) => s"Bob in $dept makes $salary per year"
  }
```

这里 depts 是一个以员工整型 ID 为键值的 Map[Int, String]，假如需要打印出 Bob 的部门和薪水，首先需要根据他的名字解析出 Bob 的 ID，然后使用这个 ID 在 depts 和 salaries 中查找。可能这是 Applicative，计算的结构已经固定下来。但这是 Monad，前面的计算结果会影响随后的计算。

> **函数式编程中的“作用”**
>
> 函数式开发者通常非正式地将类型构造器（比如 Par、Option、List、Parser、Gen 等）称为作用（effect）。它是不同于副作用的，副作用是破坏了透明引用性的。这些类型之所以被称为作用（effect）是因为它们赋予了普通的值额外的能力（Par 增加了定义并行计算的能力，Option 增加了错误的可能性，等等）。关于 effect 的使用，通常使用词语 monadic effect 和 applicative effect 去表达这个类型是一个 Monad 或 Applicative 实例。

12.3.2 对比 Parser applicative 与 Parser monad

让我们看一下另外一个例子，假如将一个逗号分隔的文件解析为两列：日期和温度。以下是一个实例文件：

```
1/1/2010, 25
2/1/2010, 28
3/1/2010, 42
4/1/2010, 53
...
```

假如事先知道文件中日期和温度是这个顺序，那么可以按这个顺序编写 Parser：

```
case class Row(date: Date, temperature: Double)

val F: Applicative[Parser] = ...
val d: Parser[Date] = ...
val temp: Parser[Double] = ...

val row: Parser[Row] = F.map2(d, temp)(Row(_, _))
val rows: Parser[List[Row]] = row.sep("\n")
```

假如不知道这个顺序需要从文件头中解析这个信息，那么需要 flatMap。这里是一个文件实例，此时列是以相反的顺序出现的：

```
# Temperature, Date
25, 1/1/2010
28, 2/1/2010
42, 3/1/2010
53, 4/1/2010
...
```

为了解析这种格式，需要基于文件头的解析（第一行以 # 开始）动态地选择我们的行解析，这里需要 flatMap：

```
case class Row(date: Date, temperature: Double)

val F: Monad[Parser] = ...
val d: Parser[Date] = ...
val temp: Parser[Double] = ...

val header: Parser[Parser[Row]] = ...
val rows: Parser[List[Row]] =
  F.flatMap (header) { row => row.sep("\n") }
```

这里解析文件头的结果是 Parser[Row]。接着使用这个 parser 去解析随后的行。因为不会提前知道列的顺序，所以这里基于文件头的解析动态地选择 Row parser。

有很多方式去陈述 Applicative 和 Monad 的不同。当然，类型的签名已经表达了它们的不同，然而还是有一些其他方式去表达不同。

- Applicative 计算有固定的结构和简单的顺序 effect，而 monadic 计算可以基于之前的计算结果动态地选择结构。
- Applicative 构建了上下文自由的计算，而 Monad 允许上下文敏感的计算。[3]
- Monad 将 effect 变成一等公民，它们可以在解释的时间被生成，而不是由程序提前选择。在 Parser 的例子中 Parser[Row] 作为解析行为的一部分被生成，并被后续的解析程序使用。

12.4 可应用函子的优势

可应用函子之所以重要是因为：

- 一般来说，实现一个像 traverse 的组合子需要的假设越少越好。相比提供 flatMap、map2 要简单一些。不然的话必须为每一个 Applicative 编写一个新的 traverse。接下来将看到这类例子。
- 因为 Applicative 相比 Monad “弱”一些，这就给了它更多的灵活性。比如拿 parsing 为例，假如不需要求助于 flatMap 去实现一个 parser，这意味着语法结构在开始解析前就决定了。因此，我们的解释器或 parser 的执行者会有更多的信息去提前知道将要做什么，可以自由地做出一些额外的假设和可能以更有效的实现方式去执行 parser。增加 flatMap 是强大的，但是这意味着需要动态地生成 parser。这样解释器可能会有更多的限制，强大是有代价的。本章接下来会有关于这个问题的更多讨论。
- Applicative functor 可组合，但是一般 monad 不可以。随后会看到这一点。

3 比如，monadic 解析器容许上下文敏感的语法，而 applicative 解析器只能处理上下文无关的语法。

12.4.1 不是所有的可应用函子都是 Monad

让我们看看两个数据类型，它们是可应用函子但不是 monad。这样的例子肯定不仅仅只有这两个。如果你做更多的函数式编程，你将毫无疑问地发现或创建许多数据类型是 applicative 并不是 monadic。[4]

可应用 Stream

第一个示例是 Stream（可能是无限的）。可以为 Stream 定义 `map2` 和 `unit`，但不是 `flatMap`。

```
val streamApplicative = new Applicative[Stream] {

  def unit[A](a: => A): Stream[A] =
    Stream.continually(a)  ←——无限的常量流。

  def map2[A,B,C](a: Stream[A], b: Stream[B])(  ←——逐个组合相应的元素。
                  f: (A,B) => C): Stream[C] =
    a zip b map f.tupled
}
```

`Applicative` 背后的方法是通过 zipping 联合相应的元素。

◈ **练习 12.4**

难：`streamApplicative.sequence` 意味着什么？`Stream` 特定的 `sequence` 的签名如下：

```
def sequence[A](a: List[Stream[A]]): Stream[List[A]]
```

校验：用 Either 变量累计错误

在第 4 章中查看了 `Either` 数据结构并考虑了如何修改它去满足报告多个错误。接下来举个具体的例子，验证一个 web 页面的提交。如果仅仅是报告第一个错误，这就意味着用户需要不断地重复提交一个页面并且每次只能修复一个错误。

这是以 monadic 方式使用 `Either` 的一个情况。首先我们为部分应用的 `Either` 类型写一个 monad。

◈ **练习 12.5**

为 `Either` 写一个 monad 实例。

```
def eitherMonad[E]: Monad[({type f[x] = Either[E, x]})#f]
```

现在下面将会发生什么，这里使用一个 `flatMap` 的序列进行调用，其中每个函数 `validName`、`validBirthdate` 和 `validPhone` 都返回类型 `Either[String, T]`。

4 Monadic 是 monad 的形容词。

```
validName(field1) flatMap (f1 =>
validBirthdate(field2) flatMap (f2 =>
validPhone(field3) map (f3 => WebForm(f1, f2, f3))
```

假如 validName 失败并返回一个错误，那么 validBirthdate 和 validPhone 就不会被执行，使用 flatMap 的计算内在地建立了一个依赖的调用链。变量 f1 只会在 validName 成功返回后才会被绑定在下一个调用。现在考虑一下同样的事情，如果使用 map3：

```
map3(
    validName(field1),
    validBirthdate(field2),
    validPhone(field3))(
    WebForm(_,_,_))
```

这里传给 map3 的三个表达式直接没有依赖关系，原则上可以想象从 Either 中收集任何 error 并构建一个 error 列表。假如使用 flatMap 而不是 map3 去实现这个，程序将会在第一个 error 处停止。

现在创建一个新的数据类型 Validation，它和 Either 十分像，除了可以显式地处理多个 error。

```
sealed trait Validation[+E, +A]

case class Failure[E](head: E, tail: Vector[E] = Vector())
    extends Validation[E, Nothing]

case class Success[A](a: A) extends Validation[Nothing, A]
```

◈ 练习 12.6

为 Validation 写一个 Applicative 实例，累计失败时的错误。注意，在失败的情况下，至少会有一个 error 存在于列表的 head，其余 error 累加在列表的 tail。

继续这个例子，考虑一个 web form 需要一个名字、一个生日和一个电话号码。

```
case class WebForm(name: String, birthdate: Date, phoneNumber: String)
```

这些数据可能以字符串的形式从用户那里收集，必须确保这些数据满足一定的描述规范。假如不满足，必须给出一个错误链表到用户去表示如何修复这个问题。这个规范可能是姓名不能为空，生日必须是 "yyyy-MM-dd" 格式，phoneNumber 必须是 10 位的。

示例 12.5 校验用户在表单上的输入

```
def validName(name: String): Validation[String, String] =
    if (name != "") Success(name)
    else Failure("Name cannot be empty")

def validBirthdate(birthdate: String): Validation[String, Date] =
    try {
        import java.text._
```

```
      Success((new SimpleDateFormat("yyyy-MM-dd")).parse(birthdate))
    } catch {
      Failure("Birthdate must be in the form yyyy-MM-dd")
    }

def validPhone(phoneNumber: String): Validation[String, String] =
    if (phoneNumber.matches("[0-9]{10}"))
      Success(phoneNumber)
    else
      Failure("Phone number must be 10 digits")
```

为了验证整个 web 表单，只需简单地使用 map3 提升（lift）WebForm 构造子。

```
def validWebForm(name: String,
                 birthdate: String,
                 phone: String): Validation[String, WebForm] =
    map3(
      validName(name),
      validBirthdate(birthdate),
      validPhone(phone))(
      WebForm(_,_,_))
```

假如部分或所有的函数返回错误，整个 validWebForm 方法将返回所有的错误集合。

12.5 可应用法则

这部分将介绍可应用函子的法则。[5] 针对每个法则，试图使用已经介绍过的数据类型进行验证（最简单的是 Option）。

12.5.1 Left and right identity

可应用函子需要遵循什么样的法则？首先，当然希望它遵循函子法则：

```
map(v)(id) == v
map(map(v)(g))(f) == map(v)(f compose g)
```

这里隐含了可应用函子的一些别的法则，回想一下是如何使用 map2 和 unit 定义 map 的：

```
def map[B](fa: F[A])(f: A => B): F[B] =
    map2(fa, unit(()))((a, _) => f(a))
```

当然，这里的定义还是很武断的，可以将 unit 放在 map2 函数调用的左边：

```
def map[B](fa: F[A])(f: A => B): F[B] =
    map2(unit(()), fa)((_, a) => f(a))
```

前两个 Applicative 法则可以总结为考虑了函子法则的 map 的两个实现。换句话说，使用 unit 和 fa:F[A] 的 map2 的调用保持了 fa 的结构不变。我们称之为左和右等同性法则（下面第一行和第二行代码分别展示）：

5 有很多不同的方式表达 Applicative 法则，更多细节可参考本章笔记。

```
map2(unit(()), fa)((_,a) => a) == fa
map2(fa, unit(()))((a,_) => a) == fa
```

12.5.2 结合律

为了引出下一个法则，结合律（associativity），让我们查看一下 map3 的签名：

```
def map3[A,B,C,D](fa: F[A],
                  fb: F[B],
                  fc: F[C])(f: (A, B, C) => D): F[D]
```

可以使用 apply 和 unit 实现 map2，但让我们考虑使用 map2 来定义它。这样必须每次绑定两个效果，会有两种选择去实现它。第一种先绑定 fa 和 fb，然后将结果和 fc 进行绑定。第二种方法是先组合 fb 和 fc，然后将结果和 fa 进行绑定。可应用函子的结合律告诉我们两种方法得到的结果是一样的。这同样也让我们想到了 monoid 和 monad 的结合律。

```
op(a, op(b, c)) == op(op(a, b), c)
compose(f, op(g, h)) == compose(compose(f, g), h)
```

可应用函子的结合律是同样的一般法则。假如没有这个法则，需要两个版本的 map3，可能是 map3L 和 map3R，这取决于不同的分组，对别的组合子同样会需要爆炸式的方法去处理不同的分组方式。

可以用 product 来表达结合律，[6] 回忆一下 product 仅仅是使用 map2 将两个 effect 组成一对：

```
def product[A,B](fa: F[A], fb: F[B]): F[(A,B)] =
  map2(fa, fb)((_,_))
```

假如在右边嵌入对，总是可以将其转化成左边的嵌入对：

```
def assoc[A,B,C](p: (A,(B,C))): ((A,B), C) =
  p match { case (a, (b, c)) => ((a,b), c) }
```

使用 product 和 assoc 组合子，结合律法则可以表示为：

```
product(product(fa,fb),fc) == map(product(fa, product(fb,fc)))(assoc)
```

注意在 == 号左边的 product 调用是左结合的，在 == 右边是右结合的。在右边我们使用 assoc 函数将结果 map 成正确的结果。

12.5.3 Naturality of product

最后的可应用函子的法则是 *naturality*。为了展示，让我们看一个使用 Option 的简单例子。

示例 12.6 提取员工姓名和年薪

```
val F: Applicative[Option] = ...

case class Employee(name: String, id: Int)
case class Pay(rate: Double, hoursPerYear: Double)
```

6 product、map 和 unit 是 Applicative 可选的组成，你能够使用 product 和 map 实现 map2 么？

```
def format(e: Option[Employee], pay: Option[Pay]): Option[String] =
  F.map2(e, pay) { (e, pay) =>
    s"${e.name} makes ${pay.rate * pay.hoursPerYear}"
  }

val e: Option[Employee] = ...
val pay: Option[Pay] = ...
format(e, pay)
```

这里对 map2 的结果进行转化，从 Employee 中提取姓名，从 Pay 中提取年收入。但可以更简单地在调用 format 前分别应用这些转化，给予 format 调用 Option [String] 和 Option[Double]，而不是 Option[Employee] 和 Option[Pay]。这个可能是一个合理的重构，这样的话 format 不需要知道 Employee 和 Pay 数据类型的具体细节。

示例 12.7 重构 format

```
val F: Applicative[Option] = ...
def format(name: Option[String], pay: Option[Double]): Option[String] =
  F.map2(e, pay) { (e, pay) => s"$e makes $pay" }
val e: Option[Employee] = ...
val pay: Option[Pay] = ...
format(
  F.map(e)(_.name),
  F.map(pay)(pay => pay.rate * pay.hoursPerYear))
```

format现在接收Option[String]作为员工名字，而不是从Option[Employee]里提取。同样pay也是如此。

在调用 map2 前使用转化去提取姓名和薪水字段，同时期望程序有着同样的意义，这种模式是时常出现的。当使用可应用函子时，一般可以选择在使用 map2 组合值之前和之后进行转化。Naturality 法则表述为无论何种情况结果都是一致的。下面是更正式的表述：

```
map2(a,b)(productF(f,g)) == product(map(a)(f), map(b)(g))
```

这里 productF 将两个函数联合成一个函数，将接收它们的参数并返回它们的结果对：

```
def productF[I,O,I2,O2](f: I => O, g: I2 => O2): (I,I2) => (O,O2) =
  (i,i2) => (f(i), g(i2))
```

可应用法则并不深奥，就像 monad 法则，这些是让可应用函子按照我们的期望进行完整性检查。它们保证了 unit、map 和 map2 具有一致性和合理的表现。

◇ **练习 12.7**

难：证明所有的 monad 是可应用函子。如果遵循 monad 法则，则其所实现的 map2 和 map 满足可应用（applicative）法则。

◈ **练习 12.8**

就像我们对两个 monoid A 和 B 进行 product，得到 monoid (A, B)，我们可以对可应用函子进行同样的操作，实现这个 product 函数：

```
def product[G[_]](G: Applicative[G]):
  Applicative[({type f[x] = (F[x], G[x])})#f]
```

◈ 练习 12.9

难：可应用函子也可以由另一种方式合成！如果 F[_] 和 G[_] 是可应用函子，那么 F[G[_]] 也是。实现这个 compose 函数：

```
def compose[G[_]](G: Applicative[G]):
    Applicative[({type f[x] = F[G[x]]})#f]
```

◇ 练习 12.10

难：证明这种合成的可应用函子满足可应用法则。这是一个极具挑战的练习。

◇ 练习 12.11

尝试一下合成 Monad，这是不可能的，但这个尝试对你理解为什么会这样有启发。

```
def compose[G[_]](G: Monad[G]): Monad[({type f[x] = F[G[x]]})#f]
```

12.6 可遍历函子

traverse 和 sequence 函数（还有一些其他操作）不直接依赖于 flatMap 帮助我们发现了可应用函子。同样可以再次通过泛化 traverse 和 sequence 发现别的抽象。让我们再次来查看 traverse 和 sequence 的函数签名：

```
def traverse[F[_],A,B](as: List[A])(f: A => F[B]): F[List[B]]
def sequence[F[_],A](fas: List[F[A]]): F[List[A]]
```

每次当你看到一个具体的类型构造器（如 List）在一个抽象的接口（如 Applicative）中出现时，你可以问一个问题："假如我将这个类型构造器抽象会发生什么？"在第 10 章中一些不是 List 的数据类型是可折叠的。那么除了 List 的这些数据类型是否是可遍历的？当然！

◈ 练习 12.12

在 Applicative trait 中，对 Map 而非 List 实现 sequence：

```
def sequenceMap[K,V](ofa: Map[K,F[V]]): F[Map[K,V]]
```

但是为每一个可遍历的数据类型编写专门的 sequence 和遍历方法是十分烦琐的。需要一个新的接口，我们将其称为 Traverse：[7]

```
trait Traverse[F[_]] {
  def traverse[G[_]:Applicative,A,B](fa: F[A])(f: A => G[B]): G[F[B]] =
    sequence(map(fa)(f))
  def sequence[G[_]:Applicative,A](fga: F[G[A]]): G[F[A]] =
    traverse(fga)(ga => ga)
}
```

7 Traversable 的名字已经被 Scala 标准库里不相关的 trait 占用了。

这里有趣的操作是 sequence。仔细观察它的函数签名，它接收 F[G[A]] 然后交换了 F 和 G 的顺序，只要 G 是一个可应用函子。现在这是一个抽象的代数概念，我们将很快了解它的含义，但是首先让我们查看 Traverse 的一些实例。

◈ 练习 12.13

对 List、Option 和 Tree 实现 Traverse 实例。

```
case class Tree[+A](head: A, tail: List[Tree[A]])
```

现在有 List、Option、Map 和 Tree 的实例。这些泛化的 traverse/sequence 意味着什么？让我们在 sequence 的调用中简单地插入一些具体的类型，然后基于它们的函数签名可以猜测这些函数是做什么的。

- List[Option[A]] => Option[List[A]]（使用 Option 作为 Applicative 对 Traverse[List].sequence 调用）假如任何一个输入链表的元素是 None，则结果返回 None。不然，将返回包裹在 Some 里的原始链表。
- Tree[Option[A]] => Option[Tree[A]]（使用 Option 作为 Applicative 对 Traverse[Tree].sequence 调用）假如任何一个输入树的元素是 None，则结果返回 None。不然，将返回包裹在 Some 里的原始的树。
- Map[K, Par[A]] => Par[Map[K,A]]（使用 Par 作为 Applicative 对 Traverse[Map[K,_].sequence 调用）产生了一个并行的计算去并行计算 map 里的所有元素。

这表明非常多的操作可以以 sequence 和 traverse 定义为最一般的方式。这将在下节中详细描述。

Traversal 就像 fold 那样接收一些数据结构并按顺序作用函数以制造一个结果。不同的是 traverse 保存了原始的结构，而 foldMap 丢弃了结构并以一个 monoid 操作的结果取代。Tree[Option[A]] => Option[Tree[A]] 这个函数签名保持了 Tree 的结构，而不是使用 monoid 合并值。

12.7 使用 Traverse

现在来探索更多可以使用 Traverse 实现的操作。这里只会接触一些表面，如果你感兴趣，跟随本章笔记里的参考资料，可以做更多的探索。

◈ 练习 12.14

难：根据 Traverse[F] 里的 traverse 方法实现 map。这表明了 Traverse 是一个函子的扩展，traverse 函数是一个泛化的 map（因为这个原因，有时也称它为*可遍历函子*）。注意，在实现 map 过程中可以选择 Applicative[G] 来调用 traverse。

```
trait Traverse[F[_]] extends Functor[F] {

    def traverse[G[_],A,B](fa: F[A])(f: A => G[B])(
                           implicit G: Applicative[G]): G[F[B]] =
        sequence(map(fa)(f))

    def sequence[G[_],A](fga: F[G[A]])(
                         implicit G: Applicative[G]): G[F[A]] =
        traverse(fga)(ga => ga)

    def map[A,B](fa: F[A])(f: A => B): F[B] = ???
}
```

Traverse 和 Foldable 的关系是什么？答案涉及 Applicative 和 Monoid 的联系。

12.7.1 从 monoid 到可应用函子

我们已经认识到 traverse 比 map 更通用。接下来将学习 traverse 可以表达 foldMap，以及延伸到表达 foldLeft 和 foldRight！再重新回顾一下 traverse 的签名：

```
def traverse[G[_]:Applicative,A,B](fa: F[A])(f: A => G[B]): G[F[B]]
```

假如 G 是一个类型构造器 ConstInt，用于将任何类型 A 转变为 Int，这样 ConstInt[A] 丢弃了它的类型参数 A 仅仅返回 Int：

```
type ConstInt[A] = Int
```

那么在 traverse 的类型签名中，假如实例化 G 为 ConstInt，它就变成：

```
def traverse[A,B](fa: F[A])(f: A => Int): Int
```

这个看起来很像 Flodable 里的 foldMap。事实上，如果 F 是 List，那么实现的这个签名是联合 List 里面的每个元素的 f 调用结果并返回一个整型的值，同时给予一个初始值对应空 list。换句话说，仅仅需要一个 Monoid[Int]，这是很容易实现的。

事实上，这里假定的常量函子，我们可以将 Monoid 转变成 Applicative。

示例 12.8 将一个 Monoid 转变成一个 Applicative

```
type Const[M, B] = M ←——ConstInt被泛化为任何M，而不仅仅是Int。

implicit def monoidApplicative[M](M: Monoid[M]) =
    new Applicative[({ type f[x] = Const[M, x] })#f] {
        def unit[A](a: => A): M = M.zero
        def map2[A,B,C](m1: M, m2: M)(f: (A,B) => C): M = M.op(m1,m2)
    }
```

这个意味着 Traverse 可以继承 Foldable，并且可以使用 traverse 来实现 foldMap：

```
trait Traverse[F[_]] extends Functor[F] with Foldable[F] { ←——┐
    ...                                  A trait可以有多个supertrait，其中它们是被with分割的。
```

```
  def foldMap[A,M](as: F[A])(f: A => M)(mb: Monoid[M]): M =
    traverse[({type f[x] = Const[M,x]})#f,A,Nothing](
      as)(f)(monoidApplicative(mb))
}
```

Scala不能推导出部分应用的Const类型别名，因此这里提供了一个注解。

注意现在 Traverse 继承了 Foldable 和 Functor！重要的是，Foldable 自身是不能继承 Functor 的。即使对大多数可折叠的数据结构比如说 List 可以使用 fold 实现 map，但这也不是通用的。

◈ **练习 12.15**

对于这个问题：为什么不可能让 Foldable 继承函子（Functor）？请找到一个让自己满意的答案。你能想到一个不是函子的 Foldable 吗？

那么 Traverse 真正用于什么？我们已经看到了一些实际的例子，比如说将一个 parser 列表转变为一个生产 list 的 parser。但是在什么样的情况下需要抽象（generalization）？Traverse 容许我们编写什么样的通用性库？

12.7.2 带状态的遍历

State 可应用函子是一个十分强大的类型。使用一个 State 行为去遍历一个集合，可以实现复杂的遍历并保持内部状态。

为了以合适的方式部分应用 State，一些类型注解在这里是不可避免的，但是 State 的遍历已经十分普通了，以至于我们可以为它创建一个特定的方法，这样对这些类型注解的编写就是一次性的了。

```
def traverseS[S,A,B](fa: F[A])(f: A => State[S, B]): State[S, F[B]] =
  traverse[({type f[x] = State[S,x]})#f,A,B](fa)(f)(Monad.stateMonad)
```

为了展示这个，这里的一个 State 遍历每个元素增加位置标签。我们保持一个整型状态并从 0 开始，在每步时加 1。

示例 12.9 对每个元素生成序号

```
def zipWithIndex[A](ta: F[A]): F[(A,Int)] =
  traverseS(ta)((a: A) => (for {
    i <- get[Int]
    _ <- set(i + 1)
  } yield (a, i))).run(0)._1
```

这个定义对 List、Tree 和任何 traversable 的类型都适用。

继续举例子，可以保持 List[A] 的状态去将一个可遍历的函子转变为 List。

示例 12.10 将可比案例函子转为 List

```
def toList[A](fa: F[A]): List[A] =
  traverseS(fa)((a: A) => (for {
    as <- get[List[A]]
```

获得当前状态，累加的list。

```
    _  <- set(a :: as) ←——添加当前的元素，并将新的list作为新的状态。
  } yield ()).run(Nil)._2.reverse
```

这里以一个空的链表 Nil 作为初始状态，每次遍历时将元素添加到累计列表的头部。这样将构建一个反向的链表，所以在完成 state 行为后将整个列表翻转。注意 yield() 是因为除了状态外不需要返回任何其他值。

当然，代码 toList 和 zipWithIndex 基本上是相同的。事实上，大部分 State 的遍历都是这种模式：获取当前状态，计算下一个状态，设置状态和产生值。这些在下面被总结为函数。

示例 12.11　分解 mapAccum 函数

```
def mapAccum[S,A,B](fa: F[A], s: S)(f: (A, S) => (B, S)): (F[B], S) =
  traverseS(fa)((a: A) => (for {
    s1 <- get[S]
    (b, s2) = f(a, s1)
    _  <- set(s2)
  } yield b)).run(s)

override def toList[A](fa: F[A]): List[A] =
  mapAccum(fa, List[A]())((a, s) => ((), a :: s))._2.reverse

def zipWithIndex[A](fa: F[A]): F[(A, Int)] =
  mapAccum(fa, 0)((a, s) => ((a, s), s + 1))._1
```

◈ **练习 12.16**

能够把任何可遍历函子转换成一个相反的列表，这是个有趣的结果——我们只要写一个函数就可以一劳永逸地翻转任何可遍历函子！写一下这个函数，想想它对 List、Tree 和其他可遍历函子意味着什么。

```
def reverse[A](fa: F[A]): F[A]
```

它应该遵循以下法则，对任何恰当的 x 和 y 类型：

```
toList(reverse(x)) ++ toList(reverse(y)) ==
    reverse(toList(y) ++ toList(x))
```

◈ **练习 12.17**

用 mapAccum 给 Traverse 特质中的 foldLeft 方法提供一个默认的实现。

12.7.3 组合可遍历结构

Traversal 的一个自然特性是保持它参数的类型。这是它的强项也是弱点，当将两个结构绑定为一个时，这个被很好地展示了。假定 Traverse[F]，可以将一个类型 F[A] 和类型 F[B] 的值合并成一个 F[C] 吗？可以尝试使用 mapAccum 来写一个通用版本的 zip。

示例 12.12 组合两个不同的结构类型

```
def zip[A,B](fa: F[A], fb: F[B]): F[(A, B)] =
  (mapAccum(fa, toList(fb)) {
    case (a, Nil) => sys.error("zip: Incompatible shapes.")
    case (a, b :: bs) => ((a, b), bs)
  })._1
```

注意到这个版本的 zip 不可以处理不同形状的参数。比如说，假如 F 是 List，那么它将不能处理不同长度的 List。在这个实现中，fb 的长度至少要比 fa 长。假如 F 是 Tree，那么 fb 必须在每层上至少和 fa 有相同数量的分支。

我们可以对这个通用的 zip 稍微做一些修改，提供两个版本，这样两边的形状就可以独立了。

示例 12.13 更灵活的 zip 实现

```
def zipL[A,B](fa: F[A], fb: F[B]): F[(A, Option[B])] =
  (mapAccum(fa, toList(fb)) {
    case (a, Nil) => ((a, None), Nil)
    case (a, b :: bs) => ((a, Some(b)), bs)
  })._1

def zipR[A,B](fa: F[A], fb: F[B]): F[(Option[A], B)] =
  (mapAccum(fb, toList(fa)) {
    case (b, Nil) => ((None, b), Nil)
    case (b, a :: as) => ((Some(a), b), as)
  })._1
```

这个实现对 List 和其他的顺序数据类型都是适用的。对于 List，zipR 的结果是保持 fb 参数的形状，假如 fb 比 fa 长将在左边填充 None。

对于更有趣的结构比如 Tree，这个实现可能不是我们想要的。在 zipL 中，它简单地将右参数打平成一个 List[B] 并丢弃了它的结构。对于 Tree 来说，将对一个树的前序遍历列表和一个 Tree 进行合并，并且不会跳过不匹配的子树。对树而言，只有当其形状一样时 zipR 和 zipL 才有更多的意义。

12.7.4 遍历融合

在第 5 章中讨论了如何将多次的结构遍历融合成一次。在第 10 章看到了如何使用 monoid products 在一次遍历中对一个可折叠的结构执行多重的计算。使用可应用函子中的 products 同样可以对一个可遍历的结构进行多重遍历的融合。

◈ 练习 12.18

使用可应用函子 product 实现两个遍历的融合。这个函数将给出两个函数 f 和 g，只遍历 fa 一次，收集这两个函数的结果。

```
def fuse[G[_],H[_],A,B](fa: F[A])(f: A => G[B], g: A => H[B])
                       (G: Applicative[G], H: Applicative[H]):
                       (G[F[B]], H[F[B]])
```

12.7.5 嵌套遍历

不仅可以组合可应用函子进行遍历融合，可遍历函子自身也是可组合的。假如有一个内嵌的结构，如 `Map[K,Option[List[V]]]`，那么可以同时遍历 map、option 和 list，并且方便地得到内部 V 的值，这是因为 `Map`、`Option` 和 `List` 都是可遍历的。

◈ **练习 12.19**

实现对两个 `Traverse` 实例的组合。

```
def compose[G[_]](implicit G: Traverse[G]):
    Traverse[({type f[x] = F[G[x]]})#f]
```

12.7.6 Monad 组合

现在来看看 monad 组合的问题。本章前面提及 `Applicative` 实例总是可以组合的，但 `Monad` 实例则不是。假如你在尝试实现通用的 monad 组合，那么将发现为了给嵌入的 monad F 和 G 实现 `join`，必须编写这样的类型 `F[G[F[G[A]]]] => F[G[A]]`。这个是不可以被表达成通用的实现的。但如果 G 碰巧是一个 `Traverse` 的实例，可以依次将 `G[F[_]]` 转换为 `F[G[_]]`，这样就生成了 `F[F[G[G[A]]]]`。那样就可以使用各自的 `Monad` 实例去 `join` 相关联的 F 层和相关联的 G 层。

◇ **练习 12.20**

难：对两个 monad 实现组合，其中的一个是可遍历的。

```
def composeM[F[_],G[_]](F: Monad[F], G: Monad[G], T: Traverse[G]):
    Monad[({type f[x] = F[G[x]]})#f]
```

表达性和强大往往是以组合性和模块性为代价的。组合 monad 的问题往往是通过编写定制化的可组合的 monad 来解决的。这类问题我们称之为 monad 转换器。比如，`OptionT` monad 转换器将 `Option` 和其他 monad 进行组合。

```
case class OptionT[M[_],A](value: M[Option[A]])(implicit M: Monad[M]) {
    def flatMap[B](f: A => OptionT[M, B]): OptionT[M, B] =
        OptionT(value flatMap {
            case None => M.unit(None)
            case Some(a) => f(a).value
        })
}
```

Option被添加到monad M的内部。

FlatMap 的定义对 M 和 Option 进行 map，并将结构 M[Option[M[Option[A]]]] 打平成 M[Option[A]]。但是这只是对 Option 的特别的实现。采用 Traverse 的通用的策略只对可遍历的函子有效。组合 State（不是可遍历的）需要编写特别的 StateT monad 转化器。没有通用的组合策略适用于每一个 monad。

本章的备注中会有更多 monad 转换器的信息。

12.8 小结

本章通过简单地对已存的 Monad 接口使用探索了两个有用的抽象，Applicative 和 Traverse。可应用函子相比 monad 没有那么昂贵，但有更好的抽象组合能力。函数 unit 和 map 容许我们 lift 值和函数，而且 map2 和 apply 让我们可以 lift 函数到更高层。可遍历函子是多次出现的 sequence 和 traverse 的抽象。放在一起，Applicative 和 Traverse 通过简单的元素构建了复杂和并行的 traversal，并且这只需要实现一次。当编写更多的函数代码时，将发现这些抽象，而且在程序中更好地利用它们。

这是第三部分的最后一章，除了 Monad、Applicative 和 Traverse 外还有很多其他抽象，可以使用这里的技术去发现新的结构。函数式程序员一直都在发现和归类这些抽象，到现在已经有非常多的抽象去表达不同的普遍模式（*arrow*、*category* 和 *comonad*，这里只罗列一些）。我们希望这些章节给了你足够的了解去自己探索这个世界。本章笔记上的一些参考资料是一个很好的开始。

第四部分将要完成函数式编程的故事。至此我们已经完成了库的编写，它是构成实际应用程序的核心部分，但是实际的程序最终需要和外部世界交互。第四部分将看到引用透明可以被应用到 I/O 操作或可变状态。甚至，所遇到的这些原理和模式可以让我们以组合和重用的方式去编写这类程序。

第四部分

作用与 I/O

函数式编程是一个完备的编程范式，我们能想到的所有程序都可以用它来表达，包括在当前位置修改数据，以及通过写文件或读数据库来与外界进行交互。在这一部分，我们将综合应用前三部分的内容来呈现如何用函数式编程来表达这些带作用的（effectful）程序。

在第 13 章中，我们会用 I/O monad 来实现对外部作用最直接的处理，这是在函数式语言里内嵌命令式编程语言的最简单的做法。第 14 章中，我们还会用同样的手段处理局部（local）作用和篡改（mutation）。这两章都会着重用可组合的开发方式来处理这些作用。在最后的第 15 章中，我们将开发一个用户流式 I/O 处理的库，并讨论如何编写组合化和模块化的程序去增量地处理 I/O 流。

本部分并不涉及 I/O 处理的所有技术细节，而是为你未来的学习引入基础思路，构建概念框架。毫无疑问，你肯定会遇到讨论之外的问题，但经过这 4 个部分的学习，你将能用函数式编程方法应对任何编程任务。

13 外部作用和 I/O

本章我们将应用所学的 monads 和代数数据类型扩展到处理外部作用（external effects），比如说读数据库和写文件。我们将为 I/O 开发一个 monad，并称之为 IO，这个将容许我们以纯函数的方式去处理此类外部作用。

我们将在本章中对*作用*和*副作用*做重要的区分。IO monad 直观地在一个纯函数程序中*内嵌声明式编程去处理 I/O 作用*，并保持了透明的引用替代。这里很清楚地从代码中区分出对外部影响的代码。

这里同样展示了一个关键技术，使用纯函数处理外部作用，也就是使用纯函数计算出一个对外部影响计算的*描述*，然后这个描述被另外一个*解释器*执行。本质上我们为申明式编程构建了一个嵌入的领域定制语言（EDSL）。这个强大的技术将在第四部分中使用。目的是使你可以使用自己定义的 EDSL 去描述有外部作用的程序。

13.1 分解作用

首先给出一个简单的有副作用的程序，寻找用 IO monad 解决的方式。

示例 13.1 带副作用的程序

```
case class Player(name: String, score: Int)

def contest(p1: Player, p2: Player): Unit =
    if (p1.score > p2.score)
        println(s"${p1.name} is the winner!")
    else if (p2.score > p1.score)
        println(s"${p2.name} is the winner!")
    else
        println("It's a draw.")
```

为了展示计算 winner 的纯函数结果 `contest` 函数耦合了 I/O 代码，我们可以将逻辑分解到纯函数 `winner` 中：

```
def winner(p1: Player, p2: Player): Option[Player] = ←——包含计算胜者的逻辑或者是平手。
    if (p1.score > p2.score) Some(p1)
    else if (p1.score < p2.score) Some(p2)
```

```
    else None

def contest(p1: Player, p2: Player): Unit = winner(p1, p2) match {  <——负责在控制台
    case Some(Player(name, _)) => println(s"$name is the winner!")      上声明胜者。
    case None => println("It's a draw.")
}
```

一个不纯粹的过程总是可以分解为一个纯函数和两个有副作用的过程：一个为纯函数提供输入，一个对纯函数的输出做相应的操作。在示例 13.1 中，我们从 contest 分解出纯函数 winner。从概念上讲，contest 有两个职责，计算出比赛的结果并进行展示。使用重构后的代码，winner 只有一个职责：计算 winner。Contest 方法保留了打印 winner 的结果到控制台的职责。

我们可以进一步重构这个代码。contest 函数仍然有两个职责，计算需要展示的消息，然后打印那个消息到控制台。我们同样可以重构出一个纯函数，这样可能让我们后续再决定在哪里展示结果，某种 UI 或写入到文件。现在让我们执行这个重构：

```
def winnerMsg(p: Option[Player]): String = p map {  <——负责选择合适的消息。
    case Player(name, _) => s"$name is the winner!"
} getOrElse "It's a draw."

def contest(p1: Player, p2: Player): Unit =  <——负责在控制台上打印消息。
    println(winnerMsg(winner(p1, p2)))
```

注意到有副作用的函数 println 此刻在程序的最外层，println 内部调用了纯函数表达式。

这看起来是个非常简单的例子，但同样的原理可以应用在更大更复杂的程序中，并且希望你可以看到这样的重构是十分自然的。我们没有改变程序的行为，只是将内部的细节分解成更小的函数。更深层次的东西在每个有副作用的函数里有一个纯函数等待着抽离出来。

我们可以将这深层次的东西规范一下。假设有一个非纯函数 f，类型为 A=>B，我们可以将 f 拆分成两个函数：

- 一个纯函数，类型是 A=>D，这里 D 是 f 结果的某种描述。
- 一个非纯函数，类型是 D=>B，可以认为是上述描述的解释器。

不久我们将延伸到处理 input 作用。此刻，让我们不断重复应用这个策略到程序上，每次应用它，将产生更纯的函数并将副作用推到更外层。我们可以称这些非纯的函数为“申明式的壳”，它包围着核心的纯函数。最终，我们达到像内置 println 这样的必须有副作用的函数，类型是 String=>Unit，那此时该怎么办？

13.2 一个简单的 IO 类型

即使像 println 这样的程序也不止做了一件事情。它可以以同样的方式进行重构，通过引入一个新的数据类型，这里我们称之为 IO：

```
trait IO { def run: Unit }

def PrintLine(msg: String): IO =
    new IO { def run = println(msg) }

def contest(p1: Player, p2: Player): IO =
    PrintLine(winnerMsg(winner(p1, p2)))
```

我们的 contest 函数现在是纯函数，它返回一个 IO 值，这个值简单地描述一个需要发生的动作，但是没有实际执行它。我们称 contest 有（或生成）了一个作用，或是有作用的，但是仅仅是 IO 的解释器（run）方法有副作用。现在 contest 只有一个职责，就是将程序的各部分构建在一起：winner 用来计算谁是胜者，winnerMsg 用来计算结果消息，并且 PrintLine 用来表示消息应该被打印到控制台。但是作用的解释的职责和实际控制台的操作是被 IO 的 run 方法持有的。

除了技术上满足透明引用的需求，IO 类型实际上还给我们带来什么？这是一个个人的判断。就像和别的数据类型一样，我们可以通过考虑它提供了什么样的代数特征来评估 IO 的优点。这是不是很有趣，通过它我们可以定义很多有用的操作和程序，同时法则给我们能力去推导这些更大的程序的行为？并不是，让我们看看能定义哪些操作：

```
trait IO { self => ←——self指向IO对象本身。
    def run: Unit
    def ++(io: IO): IO = new IO {
        def run = { self.run; io.run } ←——self是对外部IO对象的引用。
    }
}
object IO {
    def empty: IO = new IO { def run = () }
}
```

我们可以说的唯一的事可能是 IO 现在构成了一个 Monoid（empty 是单元标识，并且 ++ 是一个关联操作）。那么如果有一个 List[IO]，我们可以合并成一个 IO，关联性意味着我们可以左折叠或者右折叠。从它自身出发，这不是很有趣。这看起来像是让我们去延迟副作用的实际发生时间。

现在我们带你进入一个秘密：你作为一个程序员，开始去创建任何 API 用于表达你希望的计算，包括这些和你程序外部世界交互的部分。这种有趣、有用和组合的程序行为的描述发生在核心程序设计阶段。你在构建一门小语言和一个关联的解释器，这些将容许你表达不同的程序。假如你不喜欢你所创造的这个语言，改变它！可以像对待别的设计任务一样去解决它。

13.2.1 处理输入效果

如你之前看到的，某些时候当你构建一个小语言的时候，你将遇到不能表达的程序。至此我们的 IO 类型仅仅能表达“output”效果。没有方法去表达这样的 IO 计算，它在不

同的点等待外部源的输入。假如我们想编写一个程序，它可以提示用户以华式度输入温度，然后将其转换成摄氏温度并返回给用户。一个典型的申明式程序如下所示。[1]

示例 13.2 华式到摄氏温度转换的申明式程序

```
def fahrenheitToCelsius(f: Double): Double = (f - 32) * 5.0/9.0

def converter: Unit = {
    println("Enter a temperature in degrees Fahrenheit: ")
    val d = readLine.toDouble
    println(fahrenheitToCelsius(d))
}
```

不幸的是，我们遇到麻烦了，假如将其转换为一个纯函数并返回一个 IO：

```
def fahrenheitToCelsius(f: Double): Double =(f - 32) * 5.0/9.0

def converter: IO = {
    val prompt: IO = PrintLine(
        "Enter a temperature in degrees Fahrenheit: ")
    // now what ???
}
```

在 Scala 中，readLine 从命令窗中捕捉行输入具有副作用的函数定义。它返回一个 String。我们可以将 readLine 的调用包装在 IO 中，但是没有地方去安置结果！我们并没有方式来表达这种效果。问题是此刻 IO 类型不能表达产生有意义结果的计算。IO 的解释器仅仅产生 Unit 的输出。是否需要放弃 IO 类型，然后使用副作用呢？当然不是，让我们通过添加类型参数扩展 IO 类型来容许输入。

```
sealed trait IO[A] { self =>
    def run: A
    def map[B](f: A => B): IO[B] =
        new IO[B] { def run = f(self.run) }
    def flatMap[B](f: A => IO[B]): IO[B] =
        new IO[B] { def run = f(self.run).run }
}
```

IO 计算现在可以返回一个有意义的值。注意因为这里添加了 map 和 flatMap 函数，IO 可以在 for-comprehensions 中使用。IO 现在组成了一个 Monad：

```
object IO extends Monad[IO] {
    def unit[A](a: => A): IO[A] = new IO[A] { def run = a }
    def flatMap[A,B](fa: IO[A])(f: A => IO[B]) = fa flatMap f
    def apply[A](a: => A): IO[A] = unit(a)
}
```

这个方法让我们可以使用如IO {...}的函数式语法构建IO。

现在可以编写我们的转换器：

```
def ReadLine: IO[String] = IO { readLine }
def PrintLine(msg: String): IO[Unit] = IO { println(msg) }
def converter: IO[Unit] = for {
```

1 这里不做任何的错误处理，它只是一个为了展示的例子。

```
    _ <- PrintLine("Enter a temperature in degrees Fahrenheit: ")
    d <- ReadLine.map(_.toDouble)
    _ <- PrintLine(fahrenheitToCelsius(d).toString)
} yield ()
```

转换器定义现在没有副作用了，它是一个有计算作用的透明引用的描述，`converter.run` 是一个实际执行并产生作用的解释器。由于 `IO` 组成了一个 `Monad`，我们可以使用所有之前编写的 monadic 组合器。下面是一些使用 `IO` 的实例：

- `val echo = ReadLine.flatMap(PrintLine)`——一个 `IO[Unit]`，从命令行读取一行并打印出来
- `val readInt = ReadLine.map(_.toInt)`——一个 `IO[Int]`，从命令行读取一行并解析成 `Int` 返回
- `val readInts = readInt ** readInt`——一个 `IO[(Int, Int)]`，从命令行读取两行并解析返回 `(Int, Int)`[2]
- `replicateM(10)(ReadLine)`——一个 `IO[List[String]]`，从命令行读取 10 行并解析返回一个 list[3]

让我们实现一个更大的交互程序，它在循环中提示用户输入，并计算输入的 factorial 结果。下面是运行的结果：

```
The Amazing Factorial REPL, v2.0
q - quit
<number> - compute the factorial of the given number
<anything else> - crash spectacularly
3
factorial: 6
7
factorial: 5040
q
```

在示例 13.3 展示的代码中使用了一些之前我们没有见过的 `Monad` 函数：`when`、`foreachM` 和 `sequence_`，这个将在注释里讨论。完整的代码可在章节关联的代码中找到。代码的细节并不那么重要，重要的是展示了如何将命令式程序嵌入到 Scala 的一个纯函数子集中。所有常见的命令式编程工具都在这里——我们可以编写 loops、执行 I/O 等。

示例 13.3 使用 doWhile 循环的命令式编程

```
def factorial(n: Int): IO[Int] = for { ←——命令式的factorial使用可变的IO引用。
    acc <- ref(1) ←——分配个可变引用。
    _ <- foreachM (1 to n toStream) (i => acc.modify(_ * i).skip) ←循环中修改引用。
    result <- acc.get ←——去引用一个活动值。
} yield result
```

2 `a ** b` 和 `map2(a,b)((_,_))` 一致，它将两个作用的结果组合成一个 pair。

3 `replicateM(3)(fa)` 和 `sequence(List(fa,fa,fa))` 一样。

```
val factorialREPL: IO[Unit] = sequence_(
    IO { println(helpstring) },
    doWhile { IO { readLine } } { line =>
      when (line != "q") { for {
        n <- factorial(line.toInt)
        _ <- IO { println("factorial: " + n) }
      } yield () }
    })
```

从补充说明中获取doWhile的定义。

额外的 monad 组合子

示例 13.3 使用了一些我们之前没有见过的 monad 组合子，它对任何 monad 都可以定义。你或许在考虑这些组合子对类型来说意味着什么，而非 IO。注意它们不是每一个都对 monadic 类型有意义（比如，forever 对 Option 有什么意义？对 Stream 呢？）。

```
def doWhile[A](a: F[A])(cond: A => F[Boolean]): F[Unit] = for {
    a1 <- a
    ok <- cond(a1)
    _ <- if (ok) doWhile(a)(cond) else unit(())
} yield ()

def forever[A,B](a: F[A]): F[B] = {
    lazy val t: F[B] = forever(a)
    a flatMap (_ => t)
}

def foldM[A,B](l: Stream[A])(z: B)(f: (B,A) => F[B]): F[B] =
    l match {
      case h #:: t => f(z,h) flatMap (z2 => foldM(t)(z2)(f))
      case _ => unit(z)
    }

def foldM_[A,B](l: Stream[A])(z: B)(f: (B,A) => F[B]): F[Unit] =
    skip { foldM(l)(z)(f) }

def foreachM[A](l: Stream[A])(f: A => F[Unit]): F[Unit] =
    foldM_(l)(())((u,a) => skip(f(a)))
```

只有cond函数返回true，一直循环重复第一个参数的作用。

无限重复参数的作用。

使用函数f折叠流，组合作用并返回结果。

同foldM一致，除了不返回结果。

对流中每个元素调用函数f并组合作用。

在 Scala 中我们不必非要按照这种方式编写代码。[4] 但是它展示了函数式编程的表达性不会受到任何限制，每个程序都可以表达为纯函数的方式，即使这个函数式程序是将申明式程序嵌入到 IO monad 中。

4 如果有一个像这样独立的非纯函数块，总可以编写执行副作用的定义并包装在 IO 中——相对于使用 for-comprehension 语法和不同的 Monad 组合器，这将更高效，语法更优美。

13.2.2 简单 IO 类型的优缺点

至此我们讨论的 IO monad 是对外部作用表达的最小展示。它之所以重要主要是因为清楚地从非纯函数中剥离了纯函数，强迫我们诚实地对待和外部世界的交互，同样也展示了之前讨论的对作用重构的益处。但是当在 IO monad 中编程时，为了解决普通的声明式程序我们遇到了很多困难，这也促使函数式程序员去寻找更多可组合的方式去描述作用程序。[5] 然而，IO monad 的确提供了一些有用的优点：

- IO 计算是普通的值。我们可以在链表中存储它们，然后传递给函数，动态地创建它们，等等。任何一般的模式都可以被包装成函数并重用。
- IO 计算作为值的改进意味着我们可以编写一个更有趣的解释器而不是简单的 run 方法。在本章随后的章节中，我们将建立一个更好的 IO 类型并勾画一个使用非阻塞的 IO 解释器。更多的是，对于不同的解释器，客户端代码比如说 converter 保持不变——我们不暴露 IO 的描述给程序员！它完全是 IO 解释器的实现细节。

原始的 IO monad 同样有一些问题：

- 很多 IO 程序将在运行时溢出（overflow）调用栈并抛出 StackOverflowError。假如你在试验中还没有遇到这个问题，那么当你使用此刻的 IO 类型编写更大的程序时，你一定会遇到这个问题。比如，在之前 factorialREPL 中不断键入数字，最终栈将溢出。
- 类型 IO[A] 的值是完全不透明的。它仅仅是一个延迟的标识——一个不需要参数的函数。当我们调用 run，我们希望它最终制造 A 类型的值，但是我们没办法检查这个程序去确定它会这么做。它可能会永远停止什么都不做，或者它可能最终有结果。没有办法告知它的行为。我们可以说它太抽象，很少可以推导其行为。我们可以使用 monadic 组合子去组合它们，或者去运行它们，但是这是所有可以做的。
- 简单的 IO 类型与并发和异步操作没有关系。至此，仅仅容许我们依次一个接一个地不透明调用 IO 动作。许多 I/O 库，比如 java.nio 包，容许非阻塞和异步的 I/O。我们的 IO 类型没有能力使用这些操作。本章结束我们将改变这个通过开发更实际的 IO monad。让我们开始解决第一个问题（调用栈溢出），它将为其他两个问题的解决提供线索。

13.3 避免栈溢出

为了更好地理解 StackOverflowError，考虑下面非常简单的程序，它可以展示这个问题：

```
val p = IO.forever(PrintLine("Still going..."))
```

5 我们将在第 15 章中看到这个例子，那时将开发数据类型描述组合的 streaming I/O。

假如对 p.run 求值，在打印几千行后它将因为 StackOverflowError 而崩溃。假如你检查调用栈，会发现 run 不断地调用自己。这个问题存在于 flatMap 的定义中：

```
def flatMap[B](f: A => IO[B]): IO[B] =
  new IO[B] { def run = f(s elf.run).run }
```

这个方法创建了一个新的 IO 对象，它的 run 的定义在调用 f 之前再次调用 run。这将不断地在栈上堆积嵌入的 run 调用并最终溢出。那么可以针对这个做些什么？

13.3.1 将一个控制流转化为数据构造子

答案是十分简单的。取代使用函数调用来让程序控制执行流程，我们显式地将控制流程定义成数据类型。比如说，取代使用 flatMap 构建一个新 IO 去运行，我们可以使用一个数据类型来描述 IO 数据类型。那么解释器就可以是尾递归的循环。当遇到一个如 FlatMap(x,k) 的构造子，它将简单地解释 x，然后使用结果调用 k。下面是实现这个想法的新 IO 类型。

示例 13.4 构建一个新的 IO 类型

```
sealed trait IO[A] {
  def flatMap[B](f: A => IO[B]): IO[B] = FlatMap(this, f)
  def map[B](f: A => B): IO[B] = flatMap(f andThen (Return(_)))
}
case class Return[A](a: A) extends IO[A]
case class Suspend[A](resume: () => A) extends IO[A]
case class FlatMap[A,B](sub: IO[A], k: A => IO[B]) extends IO[B]
```

没有其他步骤并立即返回A的纯计算，但run遇到此结构体时，它知道计算已经结束了。

计算暂停，这里resume不接收任何参数并作用产生结果。

两个步骤的组合，flatmap具化为一个数据类型而不是函数。当run遇到它时，首先会处理子计算sub并当其返回后继续计算k。

这个新的 IO 类型有三个数据构造子，代表这个数据类型解释器支持的三种不同的控制流程。Return 代表 IO 动作完成，意味着可以不需要其他任何步骤返回值。Suspend 意味着我们想要执行某些作用来产生结果。FlatMap 数据构造子让我们继续或延伸计算，这时通过使用第一个计算的结果构建第二个计算。flatMap 方法可以直接简单地调用 FlatMap 的构造子并立即返回。当解释器遇到 FlatMap(sub, k)，它解释执行 sub 并使用结果调用 k。那么 k 将继续执行程序。

我们将很快了解解释器，但先让我们使用新 IO 类型编写 printLine 例子：

```
def printLine(s: String): IO[Unit] =
  Suspend(() => Return(println(s)))

val p = IO.forever(printLine("Still going..."))
```

实际上这里创建了一个无限嵌套的结构，很像 Stream。流的头是 Function0，剩余计算是“tail”：

```
FlatMap(Suspend(() => println(s)),
        _ => FlatMap(Suspend(() => println(s)),
                     _ => FlatMap(...)))
```

这里尾递归解释器遍历结构并执行作用：

```
@annotation.tailrec def run[A](io: IO[A]): A = io match {
    case Return(a) => a                          可以直接run(f(run(x)))，但这里run不在尾部。
    case Suspend(r) => r()                       所以匹配x来查看它是什么。
    case FlatMap(x, f) => x match {
        case Return(a) => run(f(a))              这里x是suspend(r)，会调用r并返回结果传递给函数f。
        case Suspend(r) => run(f(r()))
        case FlatMap(y, g) => run(y flatMap (a => g(a) flatMap f))
    }                                            这里io是像FlatMap(FlatMap(y, g), f)的表达式，进行
}                                                向右关联使得run在尾部调用，下次迭代将对y进行匹配。
```

注意取代 `FlatMap(x,f)` 里面的 `run(f(run(x)))`（因此失去尾递归），我们对 `x` 进行模式匹配，因为它仅仅是三种情况中的一种。假如是 `Return`，我们仅仅在内部调用 `f`。假如是 `Suspend`，可以仅仅从新开始执行，在其结果上使用 `f` 调用 `FlatMap`，然后递归。但如果 `x` 自身是一个 `FlatMap` 构造子，那么我们知道 `io` 由两个 `FlatMap` 构造子嵌套构成，比如：`FlatMap(FlatMap(y,g),f)`。

为了可以继续这样运行程序，紧接着自然是查看 `y` 是否是另外一个 `FlatMap` 构造子，但是表达式的嵌套可能会有任意的深度，同时希望保持尾递归。我们从新安排这个，有效地将 `(y flatMap g) flatMap f` 转变成 `y flatMap (a => g(a) flatMap f)`。这个是 monad 的关联法则应用！那么我们可以调用 `run` 在新的表达式上并保持了尾递归。因此，当实际解释程序时，将会不断地重写成右关联的顺序集。

```
FlatMap constructors:
    FlatMap(a1, a1 =>
        FlatMap(a2, a2 =>
            FlatMap(a3, a3 =>
                ...
                FlatMap(aN, aN => Return(aN)))))
```

现在将实例 `p` 传递给 `run`，它将不再会有栈溢出，即使在无限的递归 `IO` 程序中，也不会。

我们这里做了什么？当程序在 JVM 中进行函数调用时，它将在调用栈中压入调用帧，这时为了在调用结束时知道返回的地址，这样才可以继续执行。我们将这种控制逻辑在 `IO` 数据类型中显式地描述出来。当解释 `IO` 程序时，它将决定程序是否请求使用 `Suspend(s)` 执行某些“作用”（effect），还是使用 `FlatMap(x,f)` 调用子程序。取代采用调用栈，`run` 将调用 `x()`，然后通过在结果上调用 `f` 继续。而且 `f` 无论何时都立即返回，再次将控制交给 `run`。`IO` 程序可以认为是某种与 `run` 协同执行的协程（coroutine）。[6] 它不断地制造 `Suspend` 或者 `FlatMap` 请求，每次这样做时，它将先暂停（suspend）它自己的执行并返回控制给 `run`。它实际上驱动程序向前执行。像 `run` 这样的函数有时称为 trampoline（蹦床）。这种消除栈将控制返回给一个循环的技术被称为 trampolining（蹦床弹跳）。

6 假如你对 coroutine 不熟悉，可以参考 Wikipedia page（http://mng.bz/ALiI），但它对于了解接下来的内容并不是必要的。

13.3.2 Trampolining：栈溢出的通用解决方法

IO monad 的 resume 函数不必执行副作用。至此 IO 类型是一个通用的数据结构来实现 trampolining 计算，甚至没有涉及任何 I/O。

StackOverflowError 问题见证了在 Scala 中组合函数很容易就超出调用栈的空间。这个问题很容易模拟：

```
scala> val f = (x: Int) => x
f: Int => Int = <function1>

scala> val g = List.fill(100000)(f).foldLeft(f)(_ compose _)
g: Int => Int = <function1>

scala> g(42)
java.lang.StackOverflowError
```

构建了一个组合函数g，它由100000个顺序调用的函数组成。

很小的组合都可能会失败。幸运的是，我们可以使用 IO monad 解决这个问题：

```
scala> val f: Int => IO[Int] = (x: Int) => Return(x)
f: Int => IO[Int] = <function1>

scala> val g = List.fill(100000)(f).foldLeft(f) {
     |   (a, b) => x => Suspend(() => a(x).flatMap(b))
     | }
g: Int => IO[Int] = <function1>

scala> val x1 = run(g(0))
x1: Int = 0

scala> val x2 = run(g(42))
x2: Int = 42
```

构建一个大的左嵌套的flatMap调用链。

但是这里没有任何 I/O。所以 IO 的名字就不合适了。IO 的名字其实是因为 Suspend 可能包含一个副作用的函数。但是我们需要的不是一个为了 I/O 的 monad，需要的是一个尾调用消除的 monad！让我们改变它的名字：

```
sealed trait TailRec[A] {
  def flatMap[B](f: A => TailRec[B]): TailRec[B] =
    FlatMap(this, f)
  def map[B](f: A => B): TailRec[B] =
    flatMap(f andThen (Return(_)))
}
case class Return[A](a: A) extends TailRec[A]
case class Suspend[A](resume: () => A) extends TailRec[A]
case class FlatMap[A,B](sub: TailRec[A],
                        k: A => TailRec[B]) extends TailRec[B]
```

我们可以使用 TailRec 数据结构去为任何函数 A => B 增加 trampolining 功能，只需要将返回的类型 Build 修改成 TailRec[B]。刚刚的例子中将 Int => Int 转变成 Int

=> TailRec[Int]。程序只需要修改成在函数组合[7]时使用 flatMap 和在函数调用前使用 Suspend。

使用 TailRec 比直接的函数调用要慢一些，但是好处是可预测栈的使用。[8]

13.4 一个更微妙的 IO 类型

假如使用 TailRec 作为 IO 类型，这解决了栈溢出的问题，但是其他两个 monad 问题仍然存在，一个是什么样的作用将发生是不明显的，另外一个是没有并行机制或方法去执行不阻塞当前线程的 I/O 操作。

在执行过程中，run 解释器查看 TailRec 程序，比如说 FlatMap(Suspend(s),k)，这里会执行 s()。程序把控制权返回给 run，请求执行一些作用 s，等待结果，然后将结果传递给 k（可能随后返回进一步的请求）。此时，解释器并不知道程序会有什么样的作用。这是完全不透明的。唯一可以做的事情是调用 s()。不仅是任意或不明确的副作用，解释器也没法允许执行异步操作。因为 suspension 是一个 Function0，我们只能调用并等待它完成。

如果使用第 7 章中的 Par 代替阻塞的 Function0 呢？我们称之为 Async 类型，解释器现在可以支持异步执行了。

示例 13.5 定义 Async 类型

```
sealed trait Async[A] {
    def flatMap[B](f: A => Async[B]): Async[B] =
        FlatMap(this, f)
    def map[B](f: A => B): Async[B] =
        flatMap(f andThen (Return(_)))
}

case class Return[A](a: A) extends Async[A]
case class Suspend[A](resume: Par[A]) extends Async[A]
case class FlatMap[A,B](sub: Async[A],
                        k: A => Async[B]) extends Async[B]
```

注意到 Suspend 的 resume 参数现在是 Par[A] 而不是 () => A（或者 Function0[A]）。run 的实现也做了相应的调整，它现在返回了 Par[A] 而不是 A，并且我们依赖一个单独的尾递归 step 函数来关联 FlatMap 构造子：

---

7 这是第 11 章的 Kleisli 组合。换句话说，trampolined 函数在 TailRec monad 中使用了 Kleisli 组合，而不是使用普通的函数组合。

8 TailRec 实现尾部调用没被优化，但比使用直接的调用（不要提及栈安全）要快。看起来是构建和拆除栈帧的代价比将调用包装在 Suspend 里的代价大。有不同 TailRec 的变种，我们还没有进行细节的调查——没有必要每次调用都将返回到中央控制，只需要周期性地返回以避免栈溢出。比如，可以使用 exception 实现同样的概念，参照本章代码 Throw.scala。

```
@annotation.tailrec
def step[A](async: Async[A]): Async[A] = async match {
    case FlatMap(FlatMap(x,f), g) => step(x flatMap (a => f(a) flatMap g))
    case FlatMap(Return(x), f) => step(f(x))
    case _ => async
}

def run[A](async: Async[A]): Par[A] = step(async) match {
    case Return(a) => Par.unit(a)
    case Suspend(r) => Par.flatMap(r)(a => run(a))
    case FlatMap(x, f) => x match {
        case Suspend(r) => Par.flatMap(r)(a => run(f(a)))
        case _ => sys.error("Impossible; `step` eliminates these cases")
    }
}
```

Async 数据类型现在支持异步计算了，可以使用 Suspend 构造子接收任意的 Par。这是可行的，但可以进一步抽象 Suspend 里面使用的类型构造子。这里抽象 TailRec/Async 并参数化为某个类型 F 而不是特定的 Function0 或者 Par。我们命名这个抽象类型为 Free：

```
sealed trait Free[F[_],A]
case class Return[F[_],A](a: A) extends Free[F,A]
case class Suspend[F[_],A](s: F[A]) extends Free[F,A]
case class FlatMap[F[_],A,B](s: Free[F,A],
                             f: A => Free[F,B]) extends Free[F,B]
```

Free和TailRec的不同是Free使用一个类型构造器参数化。

TailRec是当F固定为Function0时Free的一个实现。

暂停的是任意F而不只是Function0。

TailRec 和 Async 是简单的类型别名：

```
type TailRec[A] = Free[Function0,A]
type Async[A] = Free[Par,A]
```

13.4.1 合理的 monad

Return 和 FlatMap 构造子证明了这个数据类型是一个对任何类型构造器 F 的 monad，这是因为它们正是产生一个 monad 所需的操作，我们称之为 *free* monad。[9]

◈ **练习 13.1**

Free 是一个对任何类型构造器 F 的 monad。实现 Free 特质的 map 和 flatMap，并针对 Free[F,_] 给出 Monad 实例。[10]

```
def freeMonad[F[_]]: Monad[({type f[a] = Free[F,a]})#f]
```

9 “Free”在此上下文中意味着可以自由地被生成而 F 不需要具有任何 monadic 结构。更多对“free”的正式定义可以参照本章笔记。

10 注意我们必须使用在第 10 章中讨论的“type lambda”去部分应用 Free 类型构造器。

◈ 练习 13.2

针对 Free[Function0,A] 实现一个特定的尾递归解释器 runTrampoline。

```
@annotation.tailrec
def runTrampoline[A](a: Free[Function0,A]): A
```

◈ 练习 13.3

难：使用给定的 Monad[F]，对 Free[F,A] 实现一个泛化的解释器。可以用之前的 Async 解释器，对你的实现模式化，包括使用 step 尾递归函数。

```
def run[F[_],A](a: Free[F,A])(implicit F: Monad[F]): F[A]
```

Free[F,A] 意味着什么？它是一个递归的结构，包含了被 0 层或多层 F 包裹的类型 A。[11] 之所以是 monad 是因为 flatMap 接收 A 并产生多层的 F。在得到结果前，解释器必须要能够处理所有层次的 F。我们可以将这个结构和其解释器看作交互的协程，类型 F 定义了这个交互的协议。通过谨慎地选择 F，我们可以精确地控制什么样的交互容许。

13.4.2 一个支持控制台 I/O 的 monad

Function0 不是类型参数 F 最简单的可能选择，而是最少限制的选择。缺少的限制让我们无法推导出 Function0[A] 会做什么。一个更加限制性的选择是一个代数数据类型，它仅仅和控制台交互。

示例 13.6 创建 Console 类型

```
sealed trait Console[A] {
    def toPar: Par[A]  <——将Console[A]解释为Par[A]。
    def toThunk: () => A  <——将Console[A]解释为Function0[A]。
}
case object ReadLine extends Console[Option[String]] {
    def toPar = Par.lazyUnit(run)
    def toThunk = () => run

    def run: Option[String] = <——帮助函数，用于ReadLine的解释器。
        try Some(readLine())
        catch { case e: Exception => None }
}
case class PrintLine(line: String) extends Console[Unit] {
    def toPar = Par.lazyUnit(println(line))
    def toThunk = () => println(line)
}
```

11 换句话说，这是一棵叶子为类型 A 的树，并按照 F 的描述进行分支。再换句话说，它是一棵语法树，用来描述一个程序，这个程序的指令按照 F 进行描述，而 A 是自由变量。

一个 Console[A] 代表一个产生 A 的计算，但是它被限制在两个可能的形式：ReadLine（类型是 Console[Option[String]]）或 PrintLine。我们为 Console 提供了两个解释器，一个转换成 Par，另一个转换为 Function0。这些解释器的实现是很直观的。

现在可以将这个数据类型嵌入到 Free 来获得一个限制的 IO 类型，它仅仅容许控制台 I/O。使用 Free 的 Suspend 构造子：

```
object Console {
    type ConsoleIO[A] = Free[Console, A]
    def readLn: ConsoleIO[Option[String]] = Suspend(ReadLine)
    def printLn(line: String): ConsoleIO[Unit] = Suspend(PrintLine(line))
}
```

使用 Free[Console,A] 类型，或者等同的 ConsoleIO[A]，可以编写和控制台交互的程序，并且可以合理地期望它不会执行其他的 I/O：[12]

```
val f1: Free[Console, Option[String]] = for {
    _ <- printLn("I can only interact with the console.")
    ln <- readLn
} yield ln
```

注意这里不是Scala标准的readLine和println，而是之前定义的monadic方法。

这听起来很好，但是如何能够实际地运行 ConsoleIO 呢？回忆一下 run 的签名：

```
def run[F[_],A](a: Free[F,A])(implicit F: Monad[F]): F[A]
```

为了运行 Free[Console,A]，我们需要一个 Monad[Console]。注意不可能为 Console 实现 flatMap：

```
sealed trait Console[A] {
    def flatMap[B](f: A => Console[B]): Console[B] = this match {
        case ReadLine => ???
        case PrintLine(s) => ???
    }
}
```

必须转化 Console 类型到其他类型（比如 Function0 或者 Par），我们将使用如下类型来实现转化：

```
trait Translate[F[_], G[_]] { def apply[A](f: F[A]): G[A] }
type ~>[F[_], G[_]] = Translate[F,G]
val consoleToFunction0 =
    new (Console ~> Function0) { def apply[A](a: Console[A]) = a.toThunk }
val consoleToPar =
    new (Console ~> Par) { def apply[A](a: Console[A]) = a.toPar }
```

任意F[A]到G[A]的转化。

这样可以用中缀语法F~>G表达Translate[F,G]。

使用这个类型，我们可以稍微泛化早些实现的 run：

```
def runFree[F[_],G[_],A](free: Free[F,A])(t: F ~> G)(
                        implicit G: Monad[G]): G[A] =
    step(free) match {
```

12 当然，从技术上说一个 Scala 程序总是有副作用的，但我们假设程序员采用了无副作用的编程规则，因为 Scala 不能为我们保证这些。

```
      case Return(a) => G.unit(a)
      case Suspend(r) => t(r)
      case FlatMap(Suspend(r),f) => G.flatMap(t(r))(a => runFree(f(a))(t))
      case _ => sys.error("Impossible; `step` eliminates these cases")
    }
```

我们接收了一个类型 `F ~> G` 的值，并且在解释 `Free[F,A]` 程序时执行转化。现在我们能方便地实现函数 `runConsoleFunction0` 和 `runConsolePar`，用来将 `Free[Console,A]` 转化为 `Function0[A]` 或 `Par[A]`：

```
def runConsoleFunction0[A](a: Free[Console,A]): () => A =
  runFree[Console,Function0,A](a)(consoleToFunction0)

def runConsolePar[A](a: Free[Console,A]): Par[A] =
  runFree[Console,Par,A](a)(consoleToPar)
```

这个依赖于 `Monad[Function0]` 和 `Monad[Par]` 实例：

```
implicit val function0Monad = new Monad[Function0] {
  def unit[A](a: => A) = () => a
  def flatMap[A,B](a: Function0[A])(f: A => Function0[B]) =
    () => f(a())()
}
implicit val parMonad = new Monad[Par] {
  def unit[A](a: => A) = Par.unit(a)
  def flatMap[A,B](a: Par[A])(f: A => Par[B]) =
    Par.fork { Par.flatMap(a)(f) }
}
```

◇ **练习 13.4**

难：事实证明 `runConsoleFunction0` 不是栈安全的，因为 `flatMap` 对 `Function0` 不是栈安全的（跟原始代码有相同的问题，原生 `IO` 类型调用自身的 `flatMap` 实现）。用 `runFree` 进行转换，然后实现一个栈安全的 `runConsole`。

```
def translate[F[_],G[_],A](f: Free[F,A])(fg: F ~> G): Free[G,A]
def runConsole[A](a: Free[Console,A]): A
```

类型 `Free[F,A]` 的值就像使用 `F` 指令集编写的程序。在 `Console` 的例子中，两个指令是 `PrintLine` 和 `ReadLine`。`Free` 提供了递归的骨架（`Suspend`）和 monadic 变量替换（`FlatMap` 和 `Return`）。我们可以为不同指令集引入其他的 `F`，比如，不同的 I/O 能力——一个文件系统 `F` 容许对文件系统的读 / 写。或者我们能有一个网络 `F` 来打开网路并读取数据，等等。

13.4.3 纯解释器

注意到 `ConsoleIO` 类型隐含了某些作用一定发生了！这是解释器的责任。我们可以选择将 `Console` 动作转化为纯值并不执行 I/O！比如，一个测试目的的解释器可以忽略 `PrintLine` 请求并总是返回一个常量字符串。为了实现这个，我们将 `Console` 请求转

化为一个 `String => A`，这个函数在 `A` 里构建了一个 monad，正如在第 11 章练习 11.20（`readerMonad`）中所看到的那样。

示例 13.7 创建我们的 ConsoleReader 类型

```
case class ConsoleReader[A](run: String => A) {  ←一个具化的reader monad。
  def map[B](f: A => B): ConsoleReader[B] =
    ConsoleReader(r => f(run(r)))
  def flatMap[B](f: A => ConsoleReader[B]): ConsoleReader[B] =
    ConsoleReader(r => f(run(r)).run(r))
}
object ConsoleReader {
  implicit val monad = new Monad[ConsoleReader] {
    def unit[A](a: => A) = ConsoleReader(_ => a)
    def flatMap[A,B](ra: ConsoleReader[A])(f: A => ConsoleReader[B]) =
      ra flatMap f
  }
}
```

在 `Console` 里引进了另外一个函数 `toReader`，并使用它实现 `runConsoleReader`：

```
sealed trait Console[A] {
  ...
  def toReader: ConsoleReader[A]
}
val consoleToReader = new (Console ~> ConsoleReader) {
  def apply[A](a: Console[A]) = a.toReader
}

@annotation.tailrec
def runConsoleReader[A](io: ConsoleIO[A]): ConsoleReader[A] =
  runFree[Console,ConsoleReader,A](io)(consoleToReader)
```

或者为了更完整的控制台 I/O 模拟，我们可以编写一个解释器并使用两个 list，一个代表输入缓存，另一个代表输出缓存。当解释器遇到一个 `ReadLine` 时，可以从输入缓存中弹出一个元素，当遇到一个 `PrintLine(s)` 时，可以向输出缓存中压入一个元素。

```
sealed trait Console[A] {
  ...
  def toState: ConsoleState[A]
}
case class Buffers(in: List[String], out: List[String])  ←
        封装一对缓存。in缓存用于ReadLine请求的输入，out缓存将接收PrintLine请求中的字符串。
case class ConsoleState[A](run: Buffers => (A, Buffers)) { ... }  ←
                                                          具化的state动作。

object ConsoleState {
  implicit val monad: Monad[ConsoleState] = ...
}

val consoleToState = new (Console ~> ConsoleState) {
  def apply[A](a: Console[A]) = a.toState
}
```

```
                                                    转化为一个纯的state动作。
def runConsoleState[A](io: ConsoleIO[A]): ConsoleState[A] = ←┘
    runFree[Console,ConsoleState,A](io)(consoleToState)
```

这将使得我们有多个解释器，比如，能使用 runConsoleState 去基于第 8 章中提及的属性测试库进行控制台程序测试，然后在实际中使用 runConsole。[13]

事实上可以编写通用的 runFree 来将 Free 程序转变成 State 或 Reader，这是十分棒的。Free 类型不需要任何的副作用。比如从 ConsoleIO 程序来看，我们不知道（不关心）它将会被什么样的解释器执行，可以是有副作用的 runConsole 或没有副作用的 runConsoleState。我们关心的是，*程序是引用透明的表达式*——一个纯的计算。解释器可以自由选择副作用与否，它是完全独立的。

13.5 非阻塞和异步 I/O

考虑最后一个问题，使原始的 IO monad 来支持非阻塞或异步 I/O。当执行 I/O 时，我们经常需要调用的操作不占用 CPU 但很耗时。包括从服务器网络接口接收一个连接，从输入流中读取一块块字节，写入大量的字节到文件中，等等。让我们考虑如何用 Free 解释器进行实现。

当方法 runConsole 遇到一个 Suspend(s)，s 将是一个 Console 类型并会将 f 从 Console 转化为一个目标 monad。为了支持非阻塞异步 I/O，我们简单改变目标 monad 从 Function0 到 Par，或者其他的并发 monad，比如说 scala.concurrent.Future。那么就像能为 Console 编写纯的和有作用的解释器一样，我们同样可以编写阻塞和非阻塞解释器，只需要改变目标 monad。[14]

来看一个例子。这里，runConsolePar 将 Console 请求转化为 Par 动作，然后合并成 Par[A]。可以认为这是一种编译，取代抽象的 Console 请求，使用更具体的 Par 请求替代，当 Par 值运行时，它将负责从标准的输入输出中读写字节。

```
scala> def p: ConsoleIO[Unit] = for {
     |   _ <- printLn("What's your name?")
     |   n <- readLn
     |   _ <- n match{
     |     case Some(n) => printLn(s"Hello, $n!")
     |     case None => printLn(s"Fine, be that way.")
     |   }
     | } yield ()
```

---

13 注意 runConsoleReader 和 runConsoleState 不是栈安全的。同样的原因 runConsoleFunction0 不是栈安全的。可以通过改变表达修复它，对 ConsoleReader 使用 string => TailRec[A]，对 ConsoleState 使用 Buffers => TailRec[(A,Buffers)]。

14 第 7 章中的 Par monad 不处理任何的异常。参照本章解答的源码文件 Task.scala，其中异步 I/O monad 的例子合理地处理了异常。

```
p: ConsoleIO[Unit] =
FlatMap(Suspend(PrintLine(What's your name?)),<function1>)

scala> val q = runConsolePar(p)
q: Par[Unit] = <function1>
```

虽然这个简单的例子使用 Par 运行，理论上允许异步的操作，但没有使用任何异步操作，readLine 和 println 都是阻塞的 I/O 调用。有 I/O 库支持非阻塞的 I/O 调用，而且 Par 可以绑定这些库。这些库的细节不同，但一个通用的原则是，非阻塞的字节源会有以下的接口：

```
trait Source {
  def readBytes(
    numBytes: Int,
    callback: Either[Throwable, Array[Byte]] => Unit): Unit
}
```

这里假设 readBytes 会立即返回。我们给予 readBytes 一个回调函数来表明该做什么，当结果变得可用或者 I/O 系统发生错误时。

很明显直接使用这种库是非常不舒服的。[15] 我们希望代码操作一个可组合的 monadic 接口，并将底层的非阻塞 I/O 库细节进行抽象。幸运的是，Par 类型容许我们包装回调函数：

```
trait Future[+A] {
  private def apply(k: A => Unit): Unit
}

type Par[+A] = ExecutorService => Future[A]
```

Future 内部的表示和 Source 是非常相似的。都包含一个立即返回的方法，此方法同时需要一个回调函数，或者是当 A 可用时 K 被调用。将 Source.readBytes 包装在 Future 是很直观的，但需要为 Par 增加一个原始操作：[16]

```
def async[A](run: (A => Unit) => Unit): Par[A] = es => new Future {
  def apply(k: A => Unit) = run(k)
}
```

有了这个，我们可以将异步的 readBytes 以 monadic 接口 Par 的方式进行封装：

```
def nonblockingRead(source: Source, numBytes: Int):
      Par[Either[Throwable,Array[Byte]]] =
  async { (cb: Either[Throwable,Array[Byte]] => Unit) =>
    source.readBytes(numBytes, cb)
  }

def readPar(source: Source, numBytes: Int):
      Free[Par,Either[Throwable,Array[Byte]]] =
  Suspend(nonblockingRead(source, numBytes))
```

现在可以使用普通的 for 推导来构建非阻塞计算的调用链条了：

---

15 这个 API 比 Java nio 包（API at http://mng.bz/uojM）中提供的非阻塞 I/O 要优美得多。

16 这也许是 Par 中最原始的操作。第 7 章中描述的其他原始操作都可以用它实现。

```
val src: Source = ...
val prog: Free[Par,Unit] = for {
    chunk1 <- readPar(src, 1024)
    chunk2 <- readPar(src, 1024)
    ...
}
```

◇ **练习 13.5**

难：我们不打算在这里全面实现非阻塞 I/O 库，但你可能在实现自己的非阻塞库时会对 `java.nio` 这个库感兴趣（API：http://mng.bz/uojM）。首先实现一个从 `AsynchronousFileChannel`（API：http://mng.bz/X30L）中进行异步读取的方法。[17]

```
def read(file: AsynchronousFileChannel,
         fromPosition: Long,
         numBytes: Int): Par[Either[Throwable, Array[Byte]]]
```

13.6 一个通用的 IO 类型

现在可以配置一个通用的方法来编写程序去执行 I/O。对任何需要支持的 I/O 操作，可以编写一个代数的数据类型并使用 case class 代表其操作。比如说，可以使用 `Files` 数据类型表示文件 I/O，一个 `DB` 数据类型代表数据库访问，使用 `Console` 去和标准输入输出交互。对任何类型 `F`，可以生成一个 free monad `Free[F, A]` 来编写程序。它可以单独地测试并最终“编译”成底层的 I/O 类型 `Async`：

```
type IO[A] = Free[Par, A]
```

这个 `IO` 类型支持蹦床式的顺序执行（因为 `Free`）和异步的执行（因为 `Par`）。在我们主程序 main 中，将各个作用（effect）组合在这个最通用的类型下。我们所需要做的就是将给定的 `F` 转换成 `Par`。

13.6.1 最终的 main 程序

当 JVM 调用到我们的主程序时，它期望一个 `main` 方法并带有特定的签名。这个方法的返回值是 `Unit`，意味着它是有副作用的。但是我们可以委托到一个纯函数 `pureMain` 中！这样 `main` 方法仅仅需要解释执行纯函数，实际上执行作用。

示例 13.8 将副作用转变成作用

```
abstract class App {
    import java.util.concurrent._

    def unsafePerformIO[A](a: IO[A])(pool: ExecutorService): A =
        Par.run(pool)(run(a)(parMonad))
```

解释IO action，实际执行是通过将IO[A]转变为Par[A]，然后是A。方法名反映了它不是安全的调用（有副作用）。

17 这需要 Java 7 或更高的版本。

```
  def main(args: Array[String]): Unit = {
    val pool = Executors.fixedThreadPool(8)
    unsafePerformIO(pureMain(args))(pool)
  }

  def pureMain(args: IndexedSeq[String]): IO[Unit]
}
```

main所做的是解释我们的pureMain。

实际的程序从这里开始，会在App的子类中实现pureMain，它接收一个不可变的IndexedSeq做参数，而不是可变的Array。

我们想要区分副作用和作用。pureMain 程序不会有任何的副作用，它应该是可以透明引用推导的 IO[Unit] 类型。作用的执行完全被 main 包含了，它是 pureMain 的外部世界。因为程序不会察觉到这些作用的发生，但它确实发生了，这样就称之为是有作用的，但不是副作用。

13.7 为什么 IO 类型不足以支撑流式 I/O

尽管 IO monad 很灵活并且 IO 动作作为一等公民很有好处，但是 IO 类型从根本上提供了和申明式编程同层次的抽象。这意味着编写高效的、流水式的 I/O 将需要要大段的循环。

让我们看一个例子。假如需要编写一个程序来转化一个文件 fahrenheit.txt，这个文件包含华式温度的序列，按行分割，需要将其转化为一个包含摄氏温度的文件 celsius.txt。代数的描述如下所示：[18]

```
trait Files[A]
case class ReadLines(file: String) extends Files[List[String]]
case class WriteLines(file: String, lines: List[String])
  extends Files[Unit]
```

使用这个类型作为 Free[F,A] 里的类型 F，我们可以按照下面的方式编写程序：

```
val p: Free[Files,Unit] = for {
  lines <- Suspend { (ReadLines("fahrenheit.txt")) }
  cs = lines.map(s => fahrenheitToCelsius(s.toDouble).toString)
  _ <- Suspend { WriteLines("celsius.txt", cs) }
} yield ()
```

这个是可行的，尽管需要将 fahrenheit.txt 文件内容全部加载到内存中，当文件很大时会是个问题。我们期望使用基本大小固定的内存执行这个工作，读一部分内容到内存中，转化它，再写到文件中，不断地重复。为了达到这个目的，我们可能需要暴露底层的文件 API 来进行 I/O 处理：

```
trait Files[A]
case class OpenRead(file: String) extends Files[HandleR]
case class OpenWrite(file: String) extends Files[HandleW]
case class ReadLine(h: HandleR) extends Files[Option[String]]
case class WriteLine(h: HandleW, line: String) extends Files[Unit]
```

18 这个 API 中忽略了异常的处理。

```
trait HandleR
trait HandleW
```

唯一的问题就是现在需要编写一个巨大的循环：

```
def loop(f: HandleR, c: HandleW): Free[Files, Unit] = for {
  line <- Suspend { ReadLine(f) }
  _ <- line match {
    case None => IO.unit(())
    case Some(s) => Suspend {
                      WriteLine(fahrenheitToCelsius(s.toDouble))
                    } flatMap (_ => loop(f, c))
  }
} yield b

def convertFiles = for {
  f <- Suspend(OpenRead("fahrenheit.txt"))
  c <- Suspend(OpenWrite("celsius.txt"))
  _ <- loop(f,c)
} yield ()
```

编写巨大的循环没有什么本质的错误，但是它不可以组合。假如我们决定随后需要计算一个移动的 5 元素的温度平均值，修改你的循环来实现这个功能可能非常痛苦。如果和链表上的操作相比较会发现，链表可以让我们定义一个 movingAvg 的函数并且可以在转化成摄氏温度的代码之前或之后工作：

```
def movingAvg(n: Int)(l: List[Double]): List[Double]

cs = movingAvg(5)(lines.map(s => fahrenheitToCelsius(s.toDouble))).map(_.
toString)
```

即使 movingAvg 可以由更小的模块组合而成，我们仍可以使用代数的组合 windowed 来构建它：

```
def windowed[A](n: Int, l: List[A])(f: A => B)(m: Monoid[B]): List[B]
```

重点是使用一个可组合的抽象，比如说 List 进行编程要比对一个原始的 I/O 操作进行编程简单得多。List 在这里并不是很特别，它仅仅是易于使用的可组合的 API 的一个实例。同样我们不应该为了使用高效的流式的 I/O 来放弃 FP 带来的可组合特性。[19] 幸运的是我们不必这样做。这将在第 15 章看到，我们可以为执行 I/O 建造任何的抽象。假如我们喜欢 list 或 streams，可以设计与 List 相似的 API 表达 I/O 计算。如果喜欢其他组合的抽象，我们也可以发现相应的设计方式。函数式编程给予我们灵活性。

19 大家可能会问是否可以让不同的 Files 操作返回第 5 章中定义的 Stream 类型。这称为 lazy I/O。我们将在第 15 章中讨论这样做会有问题的一些原因。

13.8 小结

本章介绍了一个简单的模型以纯函数式方式去处理外部作用和 I/O。我们从重构作用（effect）开始并展示了如何将作用移除到程序的外层。通过泛化一个 IO 类型被提取出来，它让我们不需要设计副作用和外部世界进行交互。

我们所讨论的 monad 受到栈溢出问题的困扰，可以用一个被称为 *trampolining* 的通用方法解决。这也引导我们得到一个更通用的实现 *free monad*，它帮助实现了一个更强大的 IO monad 并在内部使用非阻塞异步的 I/O。

I/O monad 不是有“作用”（effect）程序的最终实现。它之所以重要是因为其代表和外部世界交互的最小公分母。实际上，因为 IO 程序较为庞大，可用性差，所以很少会被使用。在第 15 章中，我们将讨论如何建立一个更优美，组合性、可重用性更强的抽象，这将会使用和这里本质上相同的技术。

在达到这个目的之前，我们将应用所学的知识去解开其他的一些疑惑：局部作用（local effect）。这本书里不同的地方，会假设这里作用（effect）不可以观察。下一章中，我们将讨论更多细节，查看更多使用局部作用的实例，而且可以学习如何使用类型系统去限制作用访问范围。

14 本地影响和可变状态

在本书的第 1 章，我们介绍引用透明性的概念，并为纯函数式编程预设了立场，即纯函数不能改变数据或与外界进行交互。在第 13 章中，我们已经了解到这不是真的，我们是可以写出与外界交互的纯函数式和组合式程序的，而它们是由求值器间接地去影响外界的。

本章我们将开发一个更成熟的应用透明性的概念，假定它的影响被限制在一个表达式的局部，并保证程序的其余部分是感知不到的。

本章还将介绍表达式对于哪些程序是引用透明的，而哪些不是。

14.1 纯函数式的可变状态

直到现在，你可能还认为纯函数式编程是不允许使用可变状态的。其实仔细观察你会发现，在引用透明性和纯粹性的定义中并没有说不能使用局部的可变状态，看看第 1 章中我们怎么说的：

引用透明和纯粹的定义

倘若表达式 e 是引用透明的，那么对于所有的程序 p 而言，其中 e 都可以用其求值后的结果替换，且不影响 p 的原始含义。

倘若一个函数 f 是纯粹的，那么表达式 f(x) 对于所有引用透明的 x 而言也是引用透明的。

根据上面的定义，我们可以认为下面的函数是纯粹的，尽管它用到 `while`，一个在变化的 `var`，以及一个可变的数组：

示例 14.1 可变数组的 quicksort

```
def quicksort(xs: List[Int]): List[Int] = if (xs.isEmpty) xs else {
    val arr = xs.toArray
    def swap(x: Int, y: Int) = {  ←——交换数组中的两个元素。
        val tmp = arr(x)
        arr(x) = arr(y)
```

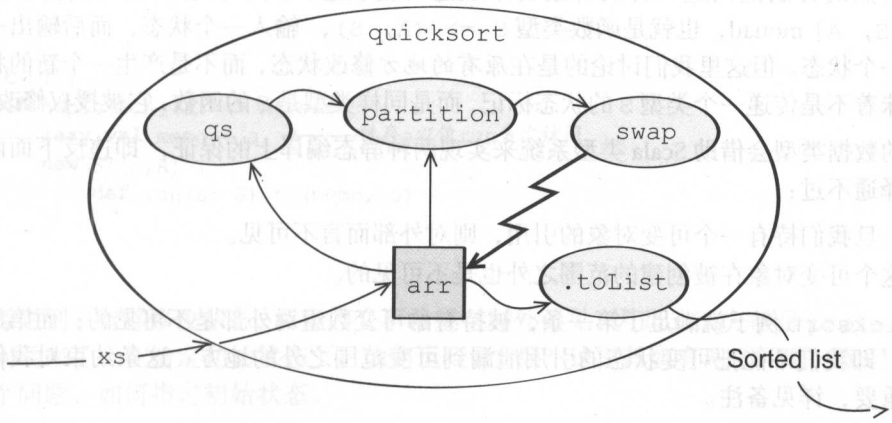
```
        arr(y) = tmp
    }

    def partition(n: Int, r: Int, pivot: Int) = {
        val pivotVal = arr(pivot)
        swap(pivot, r)
        var j = n
        for (i <- n until r) if (arr(i) < pivotVal) {
            swap(i, j)
            j += 1
        }
        swap(j, r)
        j
    }
    def qs(n: Int, r: Int): Unit = if (n < r) {
        val pi = partition(n, r, n + (r - n) / 2)
        qs(n, pi - 1)
        qs(pi + 1, r)
    }
    qs(0, arr.length - 1)
    arr.toList
}
```

将数组的元素与pivot比较，比它大的和比它小的各分一部分。

基于原数组的分治排序。

quicksort 函数首先将列表转换成一个可变数组，再采用众所周知的快排算法对可变数据数组进行排序，最后将数组又变回列表返回。在 quicksort 中，任何子表达式都不可能是引用透明的，所有本地方法 swap、partition 和 qs 也不可能是纯粹的，相反 quicksort 外部没有任何代码会保持对可变数组的引用，也就是说所有的可变性都局限在内部，那么整个函数其实是纯粹的。换言之，对于任何引用透明的 List[Int] 类型表达式 xs，quicksort(xs) 同样是引用透明的。

局部作用



只要函数以外的部分无法引用那个可变的对象，那么函数内部发生的变化也就没有副作用。

类似 quicksort 这样的一些算法，需要在原有的数据结构上进行修改以便获取最佳效率。幸运的是，我们可以通过局部化的方式来安全地修改数据。任何函数都可以在内部使用有副作用的组件而依旧提供纯粹的接口给调用者，可以毫无顾忌地在我们的程序中使用这种方法。尽管我们更倾向于使用纯函数式的组件，因为这样易于实现组合，但原则上局部地使用副作用的组件构建纯粹的函数也没有什么不好。

14.2 一种限制副作用范围的数据类型

前面一节已经说明了纯函数是可能存在副作用的，除非其数据被内化（locally scoped）。就像 quicksort 尽管修改了数组，但由于是内化的创建，它是不可能被外部观察到的。相反，若 quicksort 直接修改了输入的列表（这里它是可修改的集合），那么所有的调用者都将感知到其中的副作用了。

这种不严谨的限定作用域做法不能说不对，但有时还是需要借助 Scala 的类型系统来强制局限作用域的。在 quicksort 中每个内部函数都引用了有副作用可变的数组，对此编译器是无法给我们任何帮助的。无论是不小心触及副作用，还是意外修改状态，我们都不会得到任何警告。本节中，我们将开发一种数据类型，利用 Scala 的类型系统来限制可变的作用域。[1]

我们可以用 IO 来实现，但对于局部可变状态而言并不合适。倘若 quicksort 返回 IO[List[Int]]，那就意味着它是一个可安全执行且无副作用的 IO 行为，可这完全不是一个 IO 行为的场景。它们一个内（内化可变状态），一个外（外延之 IO），是有差别的，为此需要一个新的数据类型来表现。

14.2.1 受限可变性的语言表达

最自然的方法是构造一种局部语言来表达可变状态。关于状态的读写我们已经有了 State[S, A] monad，也就是函数类型 S => (A, S)，输入一个状态，而后输出一个结果和下一个状态。但这里我们讨论的是在原有的地方修改状态，而不是产生一个新的状态。这就意味着不是传递一个类型 S 的状态标记，而是同样类型是 S 的函数，它被授权修改状态。

新的数据类型会借助 Scala 类型系统来实现两种静态编译上的保证，即违反下面两条要求则编译通不过：

- 一旦我们持有一个可变对象的引用，则对外部而言不可见。
- 这个可变对象在被创建的范围之外也是不可见的。

quicksort 例子就满足了第一条，被持有的可变数组对外部是不可见的；而第二条要求更细，即我们不能把可变状态的引用泄漏到可变范围之外的地方。这条约束对我们的场景非常重要，详见备注。

---

1 这在性能和符号表示便捷（notational convenience）上是有代价的，不如把这看成是你处理过程中的另一种技术，也无须每次在充分利用本地变更（local mutation）时使用。

类型化可变范围的其他场景

试想一个写文件的 I/O 库。在底层中，文件被读取到一个可变的缓存区，也就是一个 `Array[Byte]`，重用它而非每次新建。从性能的角度看，I/O 库返回一个“只读”的 `Seq[Byte]` 视图，背后其实是字节数组，显然要比每次复制要高效。但这样做并不安全，调用方很有可能保持了不可变序列的引用，而在字节数据被篡改时，调用方就感知到了数据的变化。为了能够做到安全的回收，我们必须限制 `Seq[Byte]` 视图的范围，并确保调用方不能（直接或间接）保留对它的引用。在第 15 章的备注中我们还会有更多相关场景的讨论。

我们把这种新的内化影响的 monad 叫 `ST`，它可以表示 *state* thread、*state* transition、*state* token 或是 *state* tag。和 `State` monad 不同的是 `run` 方法是 `protected`，其他的基本一样。

示例 14.2 新数据类型 ST

```
sealed trait ST[S,A] { self =>
    protected def run(s: S): (A,S)
    def map[B](f: A => B): ST[S,B] = new ST[S,B] {
        def run(s: S) = {
            val (a, s1) = self.run(s)
                (f(a), s1)
        }
    }
    def flatMap[B](f: A => ST[S,B]): ST[S,B] = new ST[S,B] {
        def run(s: S) = {
            val (a, s1) = self.run(s)
            f(a).run(s1)
        }
    }
}

object ST {
    def apply[S,A](a: => A) = {
        lazy val memo = a ←——缓存a以便run多次访问。
        new ST[S,A] {
            def run(s: S) = (memo, s)
        }
    }
}
```

考虑到 `S` 能够修改状态，而我们又不希望它对外可见，因此 `run` 方法是 `protected` 的。那么问题来了，在给定初始状态下，如何运行 `ST` 呢？这里其实有两个问题，我们首先回答第一个问题，如何指定初始状态。

我们无须理解 `ST` 实现细节，重要的是我们怎么利用类型系统约束可变状态的范围。

14.2.2 一种可变引用的代数表达

我们的第一个例子就是有关可变引用的 ST monad 应用。这里依旧采用组合子库加一些基元函数（primitive）的形式，其中关于可变内存单元的应有的基元函数有：

- 分配一个新的可变单元
- 修改一个可变单元
- 读取一个可变单元

用于修改引用的数据结构只不过是一个受保护的 var 的包装器：

```
sealed trait STRef[S,A] {
    protected var cell: A
    def read: ST[S,A] = ST(cell)
    def write(a: A): ST[S,Unit] = new ST[S,Unit] {
        def run(s: S) = {
            cell = a
            ((), s)
        }
    }
}

object STRef {
    def apply[S,A](a: A): ST[S, STRef[S,A]] = ST(new STRef[S,A] {
        var cell = a
    })
}
```

STRef 的读写方法都是纯的，那是因为它们都返回了 ST。你可能已经注意到了，类型 S 并非是 cell 要更改的类型，我们甚至压根就没有用到类型 S 的值。尽管如此，为了能调用 apply 并实际执行 ST 的方法，我们也只能这样啦。因此它看起来像一种令牌，被授权去修改或访问其中的单元，除此之外别无他用。

STRef 是 sealed 的，也就意味着创建其实例的方法只能是调用 STRef 伴生对象的 apply 方法。应用一个类型 A 的值便可创建 STRef，但 apply 并没有直接返回 STRef，而是 ST[T, STref[S, A]]]，借由其 run(s: S) 来间接创建 STRef。这里尤为注意的是 ST 和 STRef 中的类型 S 是一致的。

现在我们来小试一下 ST，尽管下面看起来会有点怪，那是因为我们将类型 S 随意地定为了 Nothing：

```
for {
    r1 <- STRef[Nothing,Int](1)
    r2 <- STRef[Nothing,Int](1)
    x <- r1.read
    y <- r2.read
    _ <- r1.write(y+1)
    _ <- r2.write(x+1)
    a <- r1.read
```

```
    b <- r2.read
} yield (a,b)
```

这个小程序分配两个可变的 Int 单元，然后交换了彼此的内容并加 1，最后将二者的新值读出来。代码中我们还不能调用 run，因为它是受保护的（更不用说去传递 Nothing 类型的参数值了）。接下来我们就要试着 run 起来。

14.2.3 执行修改状态的行为

现在，你可能已经弄明白了 ST monad 的意义，即用 ST 来构建运算，其执行过程是，先分配内部的可变状态，再处理可变状态完成某项任务，最后丢弃可变状态。整个计算过程都是引用透明的，因为所有可变状态都是被局限在内部不可见的。然而我们想要一种机制保障这一点，就比如，对于 STRef 中的可变量 var 而言，Scala 的类型系统应能够保证永远无法从 ST 的行为中获得 STRef。相反则会违反不可变原则，破坏了引用透明性。

那么我们要怎么安全地运行 ST 的行为呢？首先要能区分安全与不安全，比如：

- ST[S, STRef[S, Int]]（不安全）
- ST[S, Int]（安全）

前者返回了一个可变引用，而后者刚刚相反。ST[S, Int] 类型的值，其实就是一个 Int，哪怕它计算中会牵扯到局部的可变状态。二者的不同在于 STRef 会牵扯到类型 S，而 Int 不会。

我们不允许执行 ST[S, STRef[S, A]] 的行为，那样会暴露 STRef。从更抽象的层面上讲，我们不允许执行任何 ST[S, T]，只要是 T 牵涉到 S。相反，要是 ST 不会暴露可变对象，比如 ST[S, Int]，则执行就是安全的。此外，你会发现我们压根不关心 S 到底是什么，反正我们最终是要抛弃它的。对于 S 而言行为只是一种多态的表现。

为了实现安全地执行 ST 的行为，我们将引入一个新的 trait，其行为同样体现了基于 S 的多态性：

```
trait RunnableST[A] {
  def apply[S]: ST[S,A]
}
```

它和第 13 章的 Translate 有异曲同工之处，其值是一个给定 *type* S 输出一个 ST[S, A] 类型值的函数。

前一小节中，我们选取了 Nothing 作为 S，而这里完全没有这个必要，它会在 apply 被调用时指明的：

```
val p = new RunnableST[(Int, Int)] {
  def apply[S] = for {
    r1 <- STRef(1)
    r2 <- STRef(2)
    x  <- r1.read
    y  <- r2.read
    _  <- r1.write(y+1)
```

```
        _  <- r2.write(x+1)
        a  <- r1.read
        b  <- r2.read
    } yield (a,b)
}
```

接下来，我们要实现一个 runST 的函数去调用任何 RunnableST 的 apply，无所谓 S 的类型选择。由于 RunnableST 行为的多形性（polymorphic）体现在 S 上，而调用的过程中又保证不会用到，所以传一个 () 是绝对安全的，也就是 Unit 类型！

函数 runST 必须定义在 ST 的伴生对象上，因为 run 在特质 ST 中是受保护的，除了伴生对象外，其他地方都无法访问到：

```
object ST {
    def apply[S,A](a: => A) = {
        lazy val memo = a
        new ST[S,A] {
            def run(s: S) = (memo, s)
        }
    }
    def runST[A](st: RunnableST[A]): A =
        st.apply[Unit].run(()). _1
}
```

现在我们总算可以运行之前的过程 p 了：

```
scala> val p = new RunnableST[(Int, Int)] {
     |   def apply[S] = for {
     |     r1 <- STRef(1)
     |     r2 <- STRef(2)
     |     x  <- r1.read
     |     y  <- r2.read
     |     _  <- r1.write(y+1)
     |     _  <- r2.write(x+1)
     |     a  <- r1.read
     |     b  <- r2.read
     |   } yield (a,b)
     | }
p: RunnableST[(Int, Int)] = $anon$1@e3a7d65

scala> val r = ST.runST(p)
r: (Int, Int) = (3,2)
```

runST(p) 在内部使用可变状态，却没有任何副作用，它更像是一对整形数的表达式，总是返回这对整型数，其他什么都不做。

这都不是关键，关键的是我们不可能再返回一个可变的引用了，因为我们连一个返回 STRef 的 RunnableST 都无法创建了：

```
scala> new RunnableST[STRef[Nothing,Int]] {
     |   def apply[S] = STRef(1)
     | }
<console>:17: error: type mismatch;
```

```
found    : ST[S,STRef[S,Int]]
required: ST[S,STRef[Nothing,Int]]
          def apply[S] = STRef(1)
```

这个例子中，Nothing 的使用正好说明了，在 apply 方法绑定了类型 S 的情况下，要想在 new RunnableST 指定 S 的具体类型是不可能的。

这是因为 STRef 标记与 ST 行为类型 S 一致的声明，是绕不过的限制，是由 Scala 的类型系统保证的！由此可见，你是无法从 ST 的行为中将 STRef 拿出来的，也就保证了你只能在其中创建它，并总是安全地修改引用。

通配类型的备注

使用通配类型是可以绕过上述限制的。比如使用 RunnableST[STRef[_, Int]]，则允许 STRef 不受限制：

```
scala> val ref = ST.runST(new RunnableST[STRef[_, Int]] {
     |   def apply[S] = for {
     |     r1 <- STRef(1)
     |   } yield r1
     | })
ref: STRef[_, Int] = STRef$$anonfun$apply$1$$anon$6@20e88a41
```

通配类型是 Scala 兼容 Java 类型系统的产物。幸运的是，当你有了一个 STRef[_, Int] 后，要想用它则会报错：

```
scala> new RunnableST[Int] {
     |   def apply[R] = for { x <- ref.read } yield x }
 error    : type mismatch;
 found    : ST[_$1,Int]
 required: ST[R,Int]
             def apply[R] = for { x <- ref.read } yield x }
                                    ^
```

这里报错表明了通配类型背后的具体类型只有 ref 知道，而 Scala 无法证明其具体类型就是这里的 R。因此即便借助通配类型绕过限制得到了 STRef，依旧无法通过它去改变或访问其中的状态。

14.2.4 可变数组

单值的可变引用其实并没有想象中那么有用，反倒是可变数组会显得更有价值。本节我们将定义操作可变数组的 ST，然后重新实现 quicksort。为此，我们需要先实现可变数组的基元组合子：分配、读取以及修改。

示例 14.3 一个数组的 ST monad

```
sealed abstract class STArray[S,A](implicit manifest: Manifest[A]) {
  protected def value: Array[A]
  def size: ST[S,Int] = ST(value.size)
```

Scala需要implicit Manifest来构造数组。

```
    def write(i: Int, a: A): ST[S,Unit] = new ST[S,Unit] {   ←根据索引写入值到数组。
      def run(s: S) = {
        value(i) = a
        ((), s)
      }
    }

    def read(i: Int): ST[S,A] = ST(value(i))   ←根据索引从数组读取值。

    def freeze: ST[S,List[A]] = ST(value.toList)   ←将该数组转换成不可变list。
}

object STArray {
    def apply[S,A:Manifest](sz: Int, v: A): ST[S, STArray[S,A]] =   ←构造一个给定长度的数组，并用v填满它。
      new STArray[S,A] {
        lazy val value = Array.fill(sz)(v)
      }
}
```

注意，Scala 是不能直接创建一个 A 类型的数组的，必须要一个 `Manifest[A]` 出现在隐喻的上下文中。为此，Scala 标准库提供了足够多的 manifests 来满足这种场景的需要。

和 `STRef` 一样，我们总是返回一个包装了 `ST` 的 `STArray` 和对应的 `S` 类型。因此，除了 `STArray` 自身外，从 `ST` monad 外面是不能访问到裸的 `STArray` 的。

有了这些基元函数，我们便可以实现更负责的数组函数了。

◈ 练习 14.1

为 `STArray` 添加一个组合子，可以根据一个 `Map` 来填充数组。其中 `Map` 的键表示数组的下标，而值则写入对应下标的数组中。比如，`xs.fill(Map(0 -> "a", 2 -> "b"))`，则 `xs` 的第 0 个值是 `"a"`，第 2 个值是 `"b"`。请使用已有的组合子实现。

```
def fill(xs: Map[Int,A]): ST[S,Unit]
```

不是所有的方法都只能通过现有的组合子来高效实现的。举个例子，Scala 标准库就提供高效方法来将列表变为一个数组。我们不如把这变成一个基元函数：

```
def fromList[S,A:Manifest](xs: List[A]): ST[S, STArray[S,A]] =
  ST(new STArray[S,A] {
    lazy val value = xs.toArray
  })
```

14.2.5 一个纯函数的 in-place 快排实现

`quicksort` 内的组件很容易就改成 `ST`。比如，交换元素的 `swap` 函数：

```
def swap[S](i: Int, j: Int): ST[S,Unit] = for {
  x <- read(i)
  y <- read(j)
```

```
    _ <- write(i, y)
    _ <- write(j, x)
} yield ()
```

◈ 练习 14.2

实现一个纯函数式版本的 partition 和 qs。

```
def partition[S](arr: STArray[S,Int],
                 n: Int, r: Int, pivot: Int): ST[S,Int]
def qs[S](a: STArray[S,Int], n: Int, r: Int): ST[S,Unit]
```

在所有组件都被改写后，quicksort 的实现如下所示：

```
def quicksort(xs: List[Int]): List[Int] =
    if (xs.isEmpty) xs else ST.runST(new RunnableST[List[Int]] {
      def apply[S] = for {
          arr <- STArray.fromList(xs)
          size <- arr.size
          _ <- qs(arr, 0, size - 1)
          sorted <- arr.freeze
        } yield sorted
    })
```

如你所见，ST monad 允许你对任何接收到的可变数据编写出纯函数，会由 Scala 的类型系统来保证没有不安全的方式出现。

◈ 练习 14.3

如同引用和数组一样，为 scala.collection.mutable.HashMap 实现一组最小的基元组合子。

14.3 纯粹是相对于上下文的

前面的章节中，我们说过数据被局限在一定的范围才得以让作用（effect）不可见。同理，除非持有对数据的引用，不然程序是看不到可变数据的。

可是，有些作用并非不可见，这要看是谁的视角了。下面就是这样一个例子，这类副作用出现在所有 Scala 程序中，甚至其中有被大家认为是纯函数式的：

```
scala> case class Foo(s: String)

scala> val b = Foo("hello") == Foo("hello")
b: Boolean = true

scala> val c = Foo("hello") eq Foo("hello")
c: Boolean = false
```

这里 Foo("hello") 看上去非常普通，我们一向简单认为它是一段引用透明的表达式，然而它的每一次出现，都会产生一个不同的 Foo。如果是用 == 来判断两个表达式的实例，结果是意料之中的 true；要是用 eq 来测试引用相等（源自 Java 语言里的概念），我们却得到 false。倘若我们深入探究，就会发现两次 Foo("hello") 的结果指向的并非是同一个对象。

要是我们将 Foo("hello") 的值赋予 x，然后观察 x eq x，有一个不同的结果：

```
scala> val x = Foo("hello")
x: Foo = Foo(hello)

scala> val d = x eq x
d: Boolean = true
```

对比我们之前对引用透明性的定义，你会发现每个 Scala 中的对象构造器都是有副作用的。这个作用在于一旦一个新的唯一的对象被创建于内存，其对象构造器则会返回一个引用去指向这个新的对象。

对于大多数程序而言，即便如此也无所谓，毕竟这些程序不会去检查引用是否相等，也就是说只有 eq 才会让程序观察到这类副作用的发生。既然如此，我们可以说在绝大多数程序的上下文中它是没有副作用的。

而我们之前对引用透明性的定义是没有涵盖这类情况的，也就是没有考虑上下文的场景。

一个更抽象的引用透明性的定义

如果说在程序 p 中每个出现的表达式 e 都被其求值的结果给替换，且对 p 不造成任何影响，我们可以说 e 对于 p 而言是引用透明的。

这个定义只是略微修正了一下以反映出并非所有程序都观察得到同一种作用（effect），也就意味着只要不影响 e 对于 p 的引用透明性，那么 e 对于 p 而言就是不可见的。就好比，多数程序不会察觉到调用构造器的副作用，其原因就在于它们根本不会使用 eq。

可定义上还是有点比较模糊不清，什么叫“求值”？怎么才算两个程序的作用（meaning）是一样的？

在 Scala 中，第一个问题倒是有个标准答案。求值的意思是简化到一种普通形态，对于 Scala 这种严格求值的语言来讲，表达式 e 求值到普通形态，就是赋值给一个 val：

```
val v = e
```

那么具体而言，对于程序 p，e 的引用透明性就是指，用 v 替换 e 去重写 p，却不改变程序本身的意义。

可什么才算“改变程序本身的意义”呢？或者说程序的意义又是什么？这是个哲学性的问题，我们可以有很多不同的答案，但在这里暂不细说了。[2] 从抽象的层面讲，引用透明

2 参见章节备注中的超链接，获取更多阅读内容。

性总是相对于上下文的，而上下文又决定我们编写的是哪种程序，以及想要赋予的含义。建立上下文是一种选择，以决定程序含义涉及的不同切面。

下面会有进一步的探讨。

14.3.1 副作用是什么?

前文我们谈到了 eq 方法是如何察觉到对象创建的副作用的，这儿我们就更深入地聊一下可见的行为和程序含义的话题，以及我们认为的可见性分界点在哪里。看这样一个例子，一个有副作用的方法：

```
def timesTwo(x: Int) = {
    if (x < 0) println("Got a negative number")
    x*2
}
```

若是在我们的程序里，用 2 替换掉 timesTwo(1)，它都将是一个不同的程序，即使计算的结果一致，但我们可以认为其含义已经发生了变化。这对于所有调用 timesTwo 的程序或程序等价的概念而言并不完全成立。

我们需要确定标准输出的变化是我们所关心的，也就是说这部分的变化是否关系到程序的上下文。就这个例子的具体情况而言，程序是否会观察到 timesTwo 内部的 println 副作用的产生。

显然，timesTwo 是隐藏依赖 I/O 子系统的，这意味着它会访问标准输出流。但是我们可以看到，大多数我们认为是纯函数的程序都需要访问一些 Scala 环境的底层，就像在内存里构造对象而后丢弃它们。最终，我们还是需要决定哪种作用是足够重要到需要去关心的。我们可以用 IO monad 去追踪 println 的调用，但可能我们并不想关心这个，也许只是想临时打印一些调试信息，而没有必要追踪它们。一旦程序的正确行为依赖控制台打印的内容（比如 UNIX 命令行工具），我们绝对需要追踪它们了。

由此可见，追踪作用是我们作为程序员的一种选择。这是一种价值判断，其中的取舍关系到我们如何做出选择。我们可以随心所欲，但对于上下文的引用透明，Scala 已经替我们做了标准的选择。比如，如有必要我们完全可以借助类型系统追踪内存分配，但 Scala 有自动内存的管理，我们就没有必要这么做了。

我们真正要关心的是追踪哪些决定我们程序正确性的作用。若程序的基础是读写文件，那么文件 I/O 则应该被类型系统追踪起来。同样，要是程序依赖对象的引用相等，那最好能够通过静态的办法知道。静态类型信息不仅让我们知道哪种类型的作用被涉及了，还能明示我们是否关系到给定的上下文。

本章的 ST 类型和上一章的 IO monad 应该让你领略到了类型系统追踪作用的能力了。但这远没有结束，限制你的永远只会是你的想象力和 Scala 类型的表达力。

14.4 小结

本章中，我们讨论了引用透明性的两个不同层面。

我们可以在局部的范围内安全地修改数据，尽管初看时并不符合我们对纯函数的定义，但在后面我们构建了纯函数式的接口，并在其内部修改状态，而且还借助 Scala 类型系统来保证其纯粹性。

我们还讨论了副作用其实是作为程序员或语言设计者的一种选择。每当我们说函数是纯粹的，其实我们已经选定了上下文，确立了对于程序而言什么样的两种东西是相等的，什么作用是程序含义需要考虑的。

15 流式处理与增量 I/O

我们在第四部分的引言就讲过，函数式编程是一个完整的范式。一切你能想到的程序都可以用函数式来描述，包括那些要与外界交互的程序。要是只有 IO 类型可以构建那类程序的话，想必你会失望的。IO 和 ST 就像是 Scala 中的一种嵌入式的命令语言，在 IO monad 的风格下，我们的编写与传统命令式编程差不多。

其实我们能做的远不止如此，在本章中，我们将展示如何恢复到前三部分高层次组合风格开发，哪怕是一些与外界打交道的程序。这个领域的设计空间巨大，我们的目标并非做完整的探究，而是想证明我们是可以做到的。

15.1 命令式 I/O 的问题示例

在一个简单且具体的使用场景中，我们将暴露嵌入在 IO monad 里命令式 I/O 的问题，那就是写一个程序检查一个文件的行数是否超过 40000。

为了说明本章要解决的问题，这是一个被有意设计的简单任务。在 IO monad 中使用常规的命令式代码是肯定能够完成任务的，但我们还是先来看看代码。

示例 15.1 用命令式风格计算行数

```
def linesGt40k(filename: String): IO[Boolean] = IO {
    val src = io.Source.fromFile(filename)  ←——scala.io.Source有个方便的函数用来
    try {                                          从文件这样的外部资源中读取数据。
        var count = 0
        val lines: Iterator[String] = src.getLines  ←——从Source中获得一个有状态的
        while (count <= 40000 && lines.hasNext) {          Iterator。
            lines.next  ←——隐藏在下一个动作中的副作用。
            count += 1
        }
        count > 40000
    }
    finally src.close
}
```

我们可以通过 unsafePerformIO 来执行这个 IO 行为。unsafePerformIO 是一个有副作用的方法，它接收一个 IO[A]，实际地执行预期的作用（参见 13.6.1），而后返回结果 A。

尽管代码使用的都是一些低级别的元素，while 循环、可变 Iterator，以及 var，但这也有不少好处。首先，这个过程是增量的，即整个文件并没有全部装载到内存中，而是按需进行加载。只要我们不缓存，每次在内存里的只有一行数据。另外，它还可以在知道结果后就终止。

当然代码不好的地方也是有的。其中一个就是，我们必须记得在文件用完后关闭它。看似理所当然，但有时我们会忘记，或是（时常）在 finally 代码块以外的地方关闭文件，而恰巧在此之前产生了异常，那么文件将依旧是打开的。[1] 这种情况叫*资源泄漏*。文件句柄是一种典型的稀有资源，因为操作系统在一段时间内只能开启有限数量的文件句柄。当这个任务被一个大程序用于递归扫描整个目录，并找出所有超过 40000 行的文件，则它很容易就因为打开太多文件而失败。

我们想要写*资源安全*的程序，其中文件能够在不需要再用的时候被关闭（无论是正常结束还是异常），并且不允许再次被访问。类似的资源还有 network sockets、database connection 等。直接使用 IO 同样会存在问题，它并不能确保资源被安全地关闭，此外也无法通过编译器来做到这一点。要是我们的库能做到在创建时就是资源安全的就再好不过了。

除了资源安全的问题外，这样的代码还有做得不够好的地方。它将高层的算法和底层的操作缠绕在一起，尽管要想从资源中获取元素，我们理所当然地需要处理各种异常，并在结束时关闭资源，但我们的程序并不关心这些事情。它需要关心的是计数并在其值达到 40000 时返回结果，然而这些却混杂在一些 I/O 操作之中。将算法与 I/O 操作缠绕在一起也许并不丑，只是不利于组合，使代码很难扩展。比如，考虑这些场景的变化：

- 判断文件是否有超过 40000 的非空行。
- 找到在 40000 行之前的所有行首字母拼写是 "abracadabra" 的那一行。

对于第一种情况，我们可以想象给 linesGt40k 函数传递一个 String => Boolean 的参数。而对于第二种情况，我们则需要修改循环的代码来跟踪一些状态。除了会让代码变得更丑，要保证它的正确性也变得更困难了。通常而言，追求效率就不得不写一个巨复杂的循环，而它恰恰丧失了组合性。

相比之下，对 Stream[String] 做行分析的话，则：

```
lines.zipWithIndex.exists(_._2 + 1 >= 40000)
```

很赞吧！使用 Stream，要做的只是将 _._2 + 1 >= 40000、zipWithIndex 和 exists 组合起来就好了。要是只想考虑非空的行，用 filter 就可以做到：

```
lines.filter(!_.trim.isEmpty).zipWithIndex.exists(_._2 + 1 >= 40000)
```

---

1 事实上 JVM 会在垃圾回收时关闭 InputStream（scala.io.Source 背后也是它），但这并不能保证能够在恰当的时机发生，甚至有可能不发生！这在少数 Full GC 时是会有的现象。

对于第二种情况，我们可以用 Stream 上的 indexOfSlice 函数，[2] 再结合 take（在获取 40000 行后会结束）和 map（取出每行的第一个字母）：

```
lines.filter(!_.trim.isEmpty).
      take(40000).
      map(_.head).
      indexOfSlice("abracadabra".toList)
```

我们想要在读取文件时也能这么处理，但问题是我们没有 Stream[String]。在有文件的情况下，我们只能写一个 lines 函数来返回一个 IO[Stream[String]]：

```
def lines(filename: String): IO[Stream[String]] = IO {
    val src = io.Source.fromFile(filename)
    src.getLines.toStream append { src.close; Stream.empty }
}
```

我们把这种叫*延迟 I/O*。因为它没有真正返回 Stream[String] 而是置于 IO monad 之中。一旦尝试从流中读取元素，它才会去读取文件，而且只有在我们读到文件结尾时才会关闭文件。尽管延迟 I/O 解决组合性的问题，但还有其他原因导致它是有问题的：

- 这不是资源安全的。资源（这里指文件）只有在遍历到文件末尾才被释放，可通常我们都希望尽早地结束遍历（这里 exists 将会在发现匹配后就结束遍历 Stream），而不希望有资源泄漏。
- 没有什么能够限制在文件关闭之后再次对 Stream 的访问。这会导致二选一的结果，完全取决于 Stream 是否有记住（缓存）那些元素。如果有记，则会大量消耗内存；要是没有，则会发生访问已关闭文件句柄的错误。
- 由于流的读取过程是有副作用的，要是有两个线程同时对 Stream 进行遍历会产生不可预料的后果。
- 事实上，我们完全没有必要知道 Stream[String] 背后发生了什么。它可能被赋予了一些我们无法控制的函数，而这些函数也许会将它存到一种数据结构中，很长时间也不用它。现在的用法要求我们有额外的知识，而不是将 String[String] 作为一个纯粹的值来操作，我们必须要了解它背后的本质。这对于组合而言是不利的，我们不应该去了解其类型的任何细节。

15.2 一个简单的流转换器

首先我们依照已习惯的 Stream 和 List 的高抽象的风格引入*流转换器*或*流处理器*的概念。流转换器指定了一种流到另一种流的转换方式，而流指代的是某种序列，可能是延

2 要是 indexOfSlice 的参数未能满足，则返回 -1。请参见 API 文档获取更详细的内容，或是在 REPL 中尝试这个函数。

迟产生的，或由外部提供的。几个典型的例子是文件的行流、HTTP 请求流、鼠标的点击流等。考虑一个新数据类型，Process，我们尝试着用它表达流的转换形式。[3]

示例 15.2 Process 数据类型

```
sealed trait Process[I,O]

case class Emit[I,O](
      head: O,
      tail: Process[I,O] = Halt[I,O]()) ←——这里我们用默认参数来简化Emit(xs, Halt())
    extends Process[I,O]                    为Emit(xs)。

case class Await[I,O](
      recv: Option[I] => Process[I,O])
    extends Process[I,O]

case class Halt[I,O]() extends Process[I,O]
```

Process[I, O] 可用转换 I 类型的流为 O 的流，但它并非是简单的函数 Stream[I] => Stream[O]，取而代之的是一个驱动器驱动的状态机，和一个同时接收 Process 和输入流的函数。一个 Process 会是下面三个状态中的一个，其中每个都是驱动的信号：

- Emit(head, tail) 告诉驱动器将 head 值递送给输出流，而后 tail 的部分继续由状态机处理。
- Await(recv) 请求从输入流得到下一个值，驱动器则将下一个值传递给函数 recv，一旦输入流再无元素则给 None。
- Halt 告诉驱动器暂时没有任何元素要从输入流里读取，或递送给输出流。

我们这就来看一个驱动器的例子，来诠释上面的要求。下面代码就在转换一个 Stream，我们可以将它作为 Process 的方法的实现：

```
def apply(s: Stream[I]): Stream[O] = this match {
    case Halt() => Stream()
    case Await(recv) => s match {
        case h #:: t => recv(Some(h))(t)
        case xs => recv(None)(xs) ←——Stream是空的。
    }
    case Emit(h,t) => h #:: t(s)
}
```

那么，给定 p: Process[I, O] 和 in: Stream[I]，则表达式 p(in) 会得到 Stream[O]。有趣的是 Process 并不知道输入的会是什么。既然已经写出了一个处理 Stream 的 Process，想必写个文件的 Process 也不是难事，但现在不急，我们先看看如何构造一个 Process，并在 REPL 中做些尝试。

---

3 我们有意为了简单而省去了类型参数上的协逆变，本应写成 Process[-I,+O] 的。同样的，我们也省去可避免栈溢出 trampoline 写法。更多健壮性（robust）相关的讨论还请见章节备注。

15.2.1 创建 Process

我们可以将任意函数 f: I => O 变成一个 Process[I, O]，要做的只是创建一个 Await，然后将经过 f 转换的值创建 Emit 返回：

```
def liftOne[I,O](f: I => O): Process[I,O] =
    Await {
        case Some(i) => Emit(f(i))
        case None => Halt()
    }
```

尝试在 REPL 中验证一下吧：

```
scala> val p = liftOne((x: Int) => x * 2)
p: Process[Int,Int] = Await(<function1>)

scala> val xs = p(Stream(1,2,3)).toList
xs: List[Int] = List(2)
```

如上所示，Process 仅处理了一个值就停止了。为了能够处理整个流，我们需要能够重复地执行等待到递送的循环。再写一个 repeat 组合子：

```
def repeat: Process[I,O] = {
    def go(p: Process[I,O]): Process[I,O] = p match {
        case Halt() => go(this)   <-- 如果过程是停止的则重启自身。
        case Await(recv) => Await {
            case None => recv(None)   <-- 当源截至时停止循环。
            case i => go(recv(i))
        }
        case Emit(h, t) => Emit(h, go(t))
    }
    go(this)
}
```

使用一个递归步骤代替构造一个 Halt，令重复可以永远执行下去。

进而我们可以将任何函数转变成一个 Process 以映射 Stream：

```
def lift[I,O](f: I => O): Process[I,O] = liftOne(f).repeat
```

由于 repeat 组合子会无限递归，而 Emit 又是立即求值的，所以我们绝对不能对非等待的 Process 执行这样的操作！比如 Emit(1).repeat 便会得到一个始终返回 1 的无限流。记住，Process 是一个流转换器，即便我们想要一个无限流，也应该是从一个无限流转换而来的。

```
scala> val units = Stream.continually(())
units: scala.collection.immutable.Stream[Unit] = Stream((), ?)

scala> val ones = lift((_:Unit) => 1)(units)
ones: Stream[Int] = Stream(1, ?)
```

除此之外，我们还可以插入或删除其中的元素。这里就是实现 Process 的过滤器去过滤掉不满足条件 p 的元素：

```
def filter[I](p: I => Boolean): Process[I,I] =
    Await[I,I] {
```

```
      case Some(i) if p(i) => emit(i)
      case _ => Halt()
    }.repeat
```

很简单，我们等待输入，若满足条件则递送到输出。调用 repeat 则是为了继续处理直到输入流结束为止。看看 REPL 的效果：

```
scala> val even = filter((x: Int) => x % 2 == 0)
even: Process[Int,Int] = Await(<function1>)

scala> val evens = even(Stream(1,2,3,4)).toList
evens: List[Int] = List(2, 4)
```

另一个是求和，sum：

```
def sum: Process[Double,Double] = {
    def go(acc: Double): Process[Double,Double] =
        Await {
            case Some(d) => Emit(d+acc, go(d+acc))
            case None => Halt()
        }
    go(0.0)
}
```

上面这种定义 Process 的方式是一种常用模式，用内部函数跟踪当前状态，这里的状态就是总计值。

来看看 sum 的 REPL 效果：

```
scala> val s = sum(Stream(1.0, 2.0, 3.0, 4.0)).toList
s: List[Double] = List(1.0, 3.0, 6.0, 10.0)
```

我们再写一些 Process 的组合子帮助我们以习惯的方式编程。试着做下面的练习实现它们，直到你完全掌握。

◈ 练习 15.1

实现 take 仅获取给定数量的元素然后就停止；实现 drop 丢弃给定数量的元素并将剩余的返回。类似的，将其参数换成条件函数，实现 takeWhile 和 dropWhile。

```
def take[I](n: Int): Process[I,I]

def drop[I](n: Int): Process[I,I]

def takeWhile[I](f: I => Boolean): Process[I,I]

def dropWhile[I](f: I => Boolean): Process[I,I]
```

◈ 练习 15.2

实现 count，计算元素的个数。比如 count(Stream("a", "bb", "ccc")) 会得到 Stream(1,2,3)（或是 Stream(0,1,2,3)，你来决定）。

```
def count[I]: Process[I,Int]
```

◈ 练习 15.3

实现 mean 求平均值。

```
def mean: Process[Double,Double]
```

本书已经不止一次向你展示了，通常一种模式都可以被定义为一个更抽象的函数。就像这里的 sum、count 和 mean 都有共同的模式，每个都有自己的状态，然后根据输入更新状态并输出。我们可以将这抽象为组合子，loop：

```
def loop[S,I,O](z: S)(f: (I,S) => (O,S)): Process[I,O] =
  Await((i: I) => f(i,z) match {
    case (o,s2) => emit(o, loop(s2)(f))
  })
```

◇ 练习 15.4

请用 loop 实现 sum 和 count。

15.2.2 组合和追加处理

现在我们可以通过组合 Process 值来构建更复杂的转换流。比如，给定两个 Process 值 f 和 g，实现 f 的输出作为 g 的输入。这样的操作我们将它命名为 |>（表示管道或组合），作为 Process 的函数实现。[4]接着，通过 f |> g 便可实现对 f 和 g 的组合。

◈ 练习 15.5

难：让下面的类型签名引导你实现 |>。

```
def |>[O2](p2: Process[O,O2]): Process[I,O2]
```

现在我们可以写这样的表达式 filter(_ % 2 == 0) |> lift(_ + 1)，将过滤和映射合并成一个转换。若是多个的话，就可以像管道（pipeline）一样写了。

有了这种组合的写法，我们可以很容易实现 map：

```
def map[O2](f: O => O2): Process[I,O2] = this |> lift(f)
```

你会发现 Process[I,_] 就是一个函子，要是忽略掉输入的类型参数，那 Process[I,O] 就等同于是一系列 O 的值。这让 map 的实现与 Stream 和 List 的映射过程是类似的。

事实上，大部分传统序列上的操作都可以定义在 Process 上。比如 append 一个处理到另一个上，也就是两个 process x 和 y，表达式 x ++ y 表示执行完 x 后余下的输入继

4 这样的操作可能让你想起函数的组合过程，即一个函数的输出是另一个的输入。其实 Process 和 Function1 都是一个更广泛类型，叫范畴。更多细节请见章节备注。

续执行 y。实现也很简单，将 x 的 Halt 替换成 y 就行了（这和 List 将 Nil 替换成第二个列表实现 ++ 一样）：

```
def ++(p: => Process[I,O]): Process[I,O] = this match {
    case Halt() => p
    case Emit(h, t) => Emit(h, t ++ p)
    case Await(recv) => Await(recv andThen (_ ++ p))
}
```

有了 ++，我们就可以定义 flatMap：

```
def flatMap[O2](f: O => Process[I,O2]): Process[I,O2] = this match {
    case Halt() => Halt()
    case Emit(h, t) => f(h) ++ t.flatMap(f)
    case Await(recv) => Await(recv andThen (_ flatMap f))
}
```

你肯定会问那 Process[I,_] 算不算是个 monad？当然是！要实现这个 Monad，我们需要部分应用 Process 的 I 参数类型，用上之前的小技巧：

```
def monad[I]: Monad[({ type f[x] = Process[I,x]})#f] =
    new Monad[({ type f[x] = Process[I,x]})#f] {
        def unit[O](o: => O): Process[I,O] = Emit(o)

        def flatMap[O,O2](p: Process[I,O])(
                          f: O => Process[I,O2]): Process[I,O2] = p flatMap f
}
```

unit 只是递送了一个参数就停止了，与 List 的 unit 很相似。

这个 Monad 和 List 的 Monad 思路是一样的。但它比 List 有趣的地方在于，它可以接收输入，并可对其进行映射、过滤、折叠、分组等操作。这使得 Process 能够表达任何流的转换，即便它并不了解怎么获取输入，也不知道如何输出。

◇ 练习 15.6

实现 zipWithIndex，以递送结果和其计数值。如，Process("a","b").zipWithIndex 得到 Process(("a",0),("b",1))。

◇ 练习 15.7

难：尝试用从 sum 和 count 中抽象出来的组合子实现 mean。

◇ 练习 15.8

实现 exists 有多种方式。比如 exists(_ % 2 == 0)(Stream(1,3,5,6,7)) 可以得到结果 Stream(true)（终止时给出最终结果），也可以是 Stream(false, false, false, true)（终止时给出所有结果），还可以是 Stream(false,

false, false, true, false)（不终止，立即给出结果）。注意可以复用 |>，以实现类似 trim 的组合子。

```
def exists[I](f: I => Boolean): Process[I,Boolean]
```

现在我们要想表达之前行计数的问题，只需要写 count|>exists(_ > 40000)。当然，想要增加过滤器和其他转换的处理也是很容易的。

15.2.3 处理文件

行计数的问题已经可以很容易地被解决了，只是到目前为止实现是基于纯粹的流。幸运的是，要想实现文件驱动 Process 并不难。与其产生一个 Stream，我们不如学习 List 的 foldLeft 的做法。

示例 15.3 使用 Process 处理文件

```
def processFile[A,B](f: java.io.File,
                     p: Process[String, A],
                     z: B)(g: (B, A) => B): IO[B] = IO {
  @annotation.tailrec
  def go(ss: Iterator[String], cur: Process[String, A], acc: B): B =
    cur match {
      case Halt() => acc
      case Await(recv) =>
        val next = if (ss.hasNext) recv(Some(ss.next))
                   else recv(None)
        go(ss, next, acc)
      case Emit(h, t) => go(ss, t, g(acc, h))
    }
  val s = io.Source.fromFile(f)
  try go(s.getLines, p, z)
  finally s.close
}
```

那么最后答案是：

```
processFile(f, count |> exists(_ > 40000), false)(_ || _)
```

◇ **练习 15.9**

写一个程序从一个文件中读取 Double 值的华氏温度，文件中每行一个值，对每个值进行转换成摄氏温度并写入另一个文件。你的程序需要跳过空行，以及 # 开头的行。你可以使用 toCelsius 函数：

```
def toCelsius(fahrenheit: Double): Double =
  (5.0 / 9.0) * (fahrenheit - 32.0)
```

15.3 可扩展的处理类型

现在 Process 类型隐含的假设环境或上下文中必须要有一个值的流。更进一步说，驱动处理过程的协议是固定的。Process 只可能有三种驱动指令：Halt、Emit 和 Await。你没有办法扩展这个协议，除非你定义一个全新的类型。为了使 Process 可以扩展，我们在已有的协议上进行参数化以应对驱动处理过程的问题。实现起来和第 13 章的 Free 类型一样。

示例 15.4 可扩张的 Process 类型

```
trait Process[F[_],O]

object Process {
    case class Await[F[_],A,O](
        req: F[A],
        recv: Either[Throwable, A] => Process[F,O])
    extends Process[F,O] ←——recv先用Either类型使得我们可以处理异常。

    case class Emit[F[_],O](
        head: O,
        tail: Process[F,O]) extends Process[F,O]
                                        致使停止的err可能是错误，也可能是End正常结束。
    case class Halt[F[_],O](err: Throwable) extends Process[F,O] ←——┘

    case object End extends Exception ←——这个Exception表示正常结束，用来进行流程控制。
    case object Kill extends Exception ←——这个Exception强制结束，后面就会用到。
}
```

不像 Free[F,A]，Process[F,P] 代表了外部请求使用协议 F，通过 Await 产生 O（O 表示输出）类型值的流。而 F 对于 Await 而言，就像 13 章里 F 对于 Suspend 一样。

Free 与 Process 最重要的不同在于，Process 可以多次递送值，而 Free 是会 Return 一个值。为此，Process 使用 Halt 来作为过程的终结。

为了保证处理过程中资源是安全的，也就是文件句柄或数据库链接需要被关闭，Await 中 recv 的函数接收参数 Either[Throwable, A]。一旦执行 req 时出现了错误，recv 就可以决定如何做。[5] 另外，我们还约定 End 异常表示没有更多输入了，而 Kill 表示处理被强制中断，此后需要清理任何使用过的资源。[6]

Halt 有个 Throwable 的构造器参数是用来表示原因的。这个原因可能是 End，说明输入耗尽而结束；也可能是 Kill，则说明是异常中断；还有可能是其他错误（注意 Exception 也是 Throwable 的一种）。

---

5 recv 函数应该是以蹦床跳（trampolined）的方式返回 TailRec[Process[F,O]]，从而避免栈溢出的问题，但为了保持简单，我们并未这么做。

6 这里存在一个设计选择：我们选择用异常来表示 End 和 Kill，是为了方便控制流，但我们也可以选择用 Option，如 Halt[F[_],O](err: Option[Throwable])。

新的 Process 类型比之前更为抽象，具体体现在输入类型的泛化上，怎么用它我们在 15.3.3 节中再讨论。

首先，Process 的操作与 F 类型选择是无关的，这就是意味着之前定义的那些方法均可以定义在现在的 Process 上，如 ++（追加）、map 和 filter。下面是 ++（其他函数请参见章节对应的源代码）和 onHalt 的实现。

示例 15.5 onHalt 和 ++ 函数

```
def onHalt(f: Throwable => Process[F,O]): Process[F,O] = this match {
    case Halt(e) => Try(f(e)) ←——使用Try辅助函数。
    case Emit(h, t) => Emit(h, t.onHalt(f))
    case Await(req,recv) => Await(req, recv andThen (_.onHalt(f)))
}
def ++(p: => Process[F,O]): Process[F,O] =
    this.onHalt {
        case End => p ←——正常结束则继续处理p。
        case err => Halt(err) ←——否则异常中断。
    }
```

调用 p.onHalt(f) 为了在 p 结束时对 Halt(e) 中的 e 执行 f(e)，这样做允许我们提供扩展逻辑，以根据终止的原因来做合适操作。另外，使用首字母为 T 的帮助函数 Try，可以保证 Process 求值的安全性，将捕获任何异常并转换成 Halt（如下代码），这对资源安全来说非常重要。原则上，我们更想捕获并处理所有异常，而不是丢给库的使用者处理。幸运的是，只有几个关键的组合子会产生异常，只要我们能够保证它们是异常安全的，便能保证资源安全。++ 就是这么定义的，正常结束就继续，否则异常中断。

下面是帮助函数 Try：

```
def Try[F[_],O](p: => Process[F,O]): Process[F,O] =
    try p
    catch { case e: Throwable => Halt(e) }
```

我们还将引入帮助函数 await，相比 Await 构造器能更好地满足类型推导：

```
def await[F[_],A,O](
      req: F[A])(
      recv: Either[Throwable,A] => Process[F,O]): Process[F,O] =
    Await(req, recv)
```

我们再次利用 ++ 实现 flatMap。由于不确定 f 是否会抛出异常，这里依旧要确保是异常安全，因此同样使用 Try 包装一下。除此之外，和之前的定义就非常相似了：

```
def flatMap[O2](f: O => Process[F,O2]): Process[F,O2] =
    this match {
        case Halt(err) => Halt(err)
        case Emit(o, t) => Try(f(o)) ++ t.flatMap(f)
        case Await(req,recv) =>
            Await(req, recv andThen (_ flatMap f))
    }
```

F 参数赋予了 Process 极大的灵活性，让我们看看它都能做些什么吧。

15.3.1 来源

此前，我们必须写一个独立的函数来驱动过程持续地读取文件。现在我们有更高效的办法，就是使用 Process[IO,O]。[7]

为什么说 Process[IO,O] 就是一个 O 值的来源呢，看看 Await 构造器上的 IO 被替换成 F 就知道了：

```
case class Await[A,O](
    req: IO[A],
    recv: Either[Throwable, A] => Process[IO,O]
) extends Process[IO,O]
```

因此，任何“外界”的请求都可能通过执行或 flatMap 对应的 IO 行为得到。如果行为成功返回 A，则我们调用 recv 函数，或者异常由 recv 传递给另一个过程，或是对资源做恰当的清理。请看下面简单的实现：

示例 15.6 runLog 函数

```
def runLog[O](src: Process[IO,O]): IO[IndexedSeq[O]] = IO {
    val E = java.util.concurrent.Executors.newFixedThreadPool(4)
    @annotation.tailrec
    def go(cur: Process[IO,O], acc: IndexedSeq[O]): IndexedSeq[O] =
        cur match {
            case Emit(h,t) => go(t, acc :+ h)
            case Halt(End) => acc
            case Halt(err) => throw err
            case Await(req,recv) =>
                val next =
                    try recv(Right(unsafePerformIO(req)(E)))
                    catch { case err: Throwable => recv(Left(err)) }
                go(next, acc)
        }
    try go(src, IndexedSeq())
    finally E.shutdown
}
```

这里有个遍历文件的所有行的示例过程：

```
import java.io.{BufferedReader,FileReader}

val p: Process[IO, String] =
    await(IO(new BufferedReader(new FileReader("lines.txt")))) {
        case Right(b) =>
            lazy val next: Process[IO,String] = await(IO(b.readLine)) {
                case Left(e) => await(IO(b.close))(_ => Halt(e))
                case Right(line) =>
                    if (line eq null) Halt(End)
                    else Emit(line, next)
            }
```

当强制结束或异常结束发生时，进行停止处理。

当基于BufferedReader的函数readLine返回null时，说明文件已经读完了。

7 还有其他的方法能够让这个表现的资源安全，很快我们会讨论的。

```
        }
        next
    case Left(e) => Halt(e)
}
```

运行 runLog(p) 我们便可从 lines.txt 文件中读出所有行，并输出 IO[IndexedSeq[String]]。

注意，这里无论 process 是如何结束的文件都会被关闭。在 15.3.2 中，我们还将讨论如何确保所有 process 都关闭了用过的资源，并发现一些抽象的组合子来专门做这件事。

◈ **练习 15.10**

runLog 函数可以定义得更通用一些，以运行任何可以捕获异常的 Monad（例如，第 13 章中的 Task 类型，就为 IO 类型添加这种能力）。实现这个更通用版本的 runLog。注意这里的实现是没法进行尾递归优化的，完全依赖底层的 monad 来保证栈安全。

```
trait Process[F[_],O] {
    def runLog(implicit F: MonadCatch[F]): F[IndexedSeq[O]]
    ...
}

trait MonadCatch[F[_]] extends Monad[F] {
    def attempt[A](a: F[A]): F[Either[Throwable,A]]
    def fail[A](t: Throwable): F[A]
}
```

15.3.2 保证资源安全

Process[IO,O] 可以用来实现与外部资源的交互，如文件和数据库链接，但我们必须要能够保证这些资源不被泄漏，即文件被关闭，数据库链接被释放等，哪怕是有异常发生。细数一下我们已做的，看看到底还需要再做什么。Await 构造器的 recv 参数可以处理错误，恰当地执行清理工作。所有 flatMap 和其他组合子可能抛出的异常已经被捕获，并能确保它们被优雅地传递给 recv。余下的就是，保证 recv 真正地调用必要的清理代码。

具体而言，一个大文件的所有行 lines: Process[IO, String]，作为一个*来源或生产者*，它隐含地持有了一个资源（文件句柄），恰恰是我们要确保关闭的，无论生产者如何被消费。

我们何时应该关闭文件句柄呢？是在每个程序的结尾吗？不，应该是在我们确定从 lines 读完所有数据后关闭。更具体一点就是读到最后一行，即没有更多的行后才关闭文件。由此我们得出第一条规则：*一个生产者应该在没有更多值后立即释放底层的资源，无论是因为耗尽还是异常。*

只有这些是不够的，因为过程的消费者有可能在消费中更早地决定结束。考虑一下 `runLog{ lines("name.txt") |> take(5) }`，其中 `take(5)` 会在获取五个元素后就停止了，当然也有不足五个的可能。这种情况下，我们也要确保能够释放必要的资源。但我们的 `runLog` 现在显然是做不到的，因为它压根不知道其中的一个 `Process` 会提前结束。

那么我们有了第二条规则：*任何消费过程 p 输出值的过程 d 必须确保在自身停止之前执行 p 的清理行为。*

这看上去有点绕，所幸我们只需要在一个地方解决这个问题。如何借助通用的 `Process` 类型，解决这类单一输入的过程，我们会在 15.3.3 中详细介绍。

现在总结一下，一个过程 p 可能会因为下面的情况而终止：

- 当底层来源没有更多值时，生产者穷尽而发出 `End` 信号
- 消费者在生产者穷尽之前决定结束消费时，强行终结而发出 `Kill` 信号
- 由于生产者或消费者的 `e: Throwable` 而异常中断

无论什么原因，我们希望每次都能关闭底层的资源。

有了上面的原则，我们要怎么实现呢？我们必须确保 `Await` 构造器的 `recv` 函数，在遇到 `Left` 时执行清理的行为。这就需要引入一个新的组合子，`onComplete`，以便为一个 `Process` 追加额外的逻辑，无论先前的 `Process` 是如何结束的。基本上和 `++` 一样，只有一点变化：

```
def onComplete(p: => Process[F,O]): Process[F,O] =
    this.onHalt {
        case End => p.asFinalizer
        case err => p.asFinalizer ++ Halt(err)
    }
```

和++操作一样，只是这里会始终执行p，哪怕是因为错误而停止的。

工具函数。

p 过程始终会在终止时被执行，只不过要记得将发生的错误留到最后的清理阶段。`asFinalizer` 方法会让一个“普通的” `Process` 被 `Kill` 时调用自身。尽管定义上面有点微不足道，但却能够确保 `p1.onComplete(p2)` 中的 `p2` 总是会执行，即便消费者希望提前结束。

```
def asFinalizer: Process[F,O] = this match {
    case Emit(h, t) => Emit(h, t.asFinalizer)
    case Halt(e) => Halt(e)
    case Await(req,recv) => await(req) {
        case Left(Kill) => this.asFinalizer
        case x => recv(x)
    }
}
```

将上面这些揉在一起，我们便能使用 `onComplete` 组合子创建一个资源安全的基于文件的 `Process[IO,O]`。为此，我们定义一个更抽象的组合子，叫 `resource`：

```
def resource[R,O](acquire: IO[R])(
                  use: R => Process[IO,O])(
```

```
                release: R => Process[IO,O]): Process[IO,O] =
  await[IO,R,O](acquire)(r => use(r).onComplete(release(r)))
```

练习 15.11

使用 await 去“求值”IO 结果的用法并不是只能用于 IO。实现一个通用的组合子 eval 将 F[A] 升格为仅传递（emit）F[A] 结果的 Process。再实现 eval_，升格一个 F[A] 为无传递值的 Process。值得注意的是，它们的实现都无须关心 F 的细节。

```
def eval[F[_],A](a: F[A]): Process[F,A]

def eval_[F[_],A,B](a: F[A]): Process[F,B]
```

现在 lines 的实现如下：

```
def lines(filename: String): Process[IO,String] =
  resource
    { IO(io.Source.fromFile(filename)) }
    { src =>
        lazy val iter = src.getLines // a stateful iterator
        def step = if (iter.hasNext) Some(iter.next) else None
        lazy val lines: Process[IO,String] = eval(IO(step)).flatMap {
          case None => Halt(End)
          case Some(line) => Emit(line, lines)
        }
        lines
    }
    { src => eval_ { IO(src.close) } }
```

resources 组合子使用 onComplete 以确保底层的资源被释放，无论过程是如何结束的。剩下我们需要确保的是 |> 和其他行的消费者在消费完成后能够优雅地终止。接下来我们将重定义单一输入的过程，并为泛化的 Process 类型实现 |> 组合子。

15.3.3 单一输入过程

现在我们有了资源安全的来源，但却没有任何办法转化它们。有幸的是，章节的前面我们已经引入能够表现单一输入过程的 Process 类型。为了表现 Process1[I,O]，我们弄出一个恰当的 F，让它只能用 Process 为 I 类型的元素制造请求。我们这就来看看如何做到，尽管编码看上去有点不同寻常，但也没有什么新概念：

```
case class Is[I]() {
  sealed trait f[X]
  val Get = new f[I] {}
}
```

将 trait f 定义在 Is 里是有点怪，具体来看看是怎么回事。首先 f 有个参数类型 X，且只有一个实例 Get，而它的类型参数 X 固定是为了 Is[I] 的 I，那么类型 Is[I]#f[8] 只

8 对于类型 x，则 x#foo 指代的是定义于 x 之内的类型 foo。

能是 I 类型值的请求。若是写成 Is[I]#f[A]，则 Scala 会提示错误，除非将类型 A 换成 I。由此我们可以将 Process1 定义为一个别名：

```
type Process1[I,O] = Process[Is[I]#f, O]
```

为了弄明白怎么回事，试着将 Is[I]#f 代入到 Await 中：

```
case class Await[A,O](
  req: Is[I]#f[A], recv: Either[Throwable,A] => Process[Is[I]#f,O]
) extends Process[Is[I]#f,R]
```

根据 Is[I]#f 的定义，可以得出 req 只能是它的唯一值 Get: f[I]。那么，I 和 A 必须是同一个类型，以至于 recv 必须接收一个 I 作为参数，这就意味着 Await 只能用于请求 I 的值。理解这一点很重要，如果感觉还是很迷惑，不妨在纸上借助类型替换的方式，推演整个过程。

Process1 这个别名支持所有单一输入 Process 上的操作。作为对比，我们首先引入一些帮助函数去改善一下 Process 构造器调用上的类型推导。

示例 15.7 类型推导相关的帮助函数

```
def await1[I,O](
      recv: I => Process1[I,O],
      fallback: Process1[I,O] = halt1[I,O]): Process1[I, O] =
  Await(Get[I], (e: Either[Throwable,I]) => e match {
      case Left(End) => fallback
      case Left(err) => Halt(err)
      case Right(i) => Try(recv(i))
  })

def emit1[I,O](h: O, tl: Process1[I,O] = halt1[I,O]): Process1[I,O] = emit(h, tl)

def halt1[I,O]: Process1[I,O] = Halt[Is[I]#f, O](End)
```

有了这些，像 lift 和 filter 之类的组合子看起来就更明确一些：

```
def lift[I,O](f: I => O): Process1[I,O] =
  await1[I,O](i => emit(f(i))) repeat

def filter[I](f: I => Boolean): Process1[I,I] =
  await1[I,I](i => if (f(i)) emit(i) else halt1) repeat
```

关于组合，这里的实现和之前差不多，只是我们需要保证，在右边的过程终止之前，左边的 cleanup 行为得以运行：

```
def |>[O2](p2: Process1[O,O2]): Process[F,O2] = {
  p2 match {                    在停止之前，优雅地结束this，并应用++保留前面可能发生的任何异常。
    case Halt(e) => this.kill onHalt { Halt(e) ++ Halt(e2) }  <——┘
    case Emit(h, t) => Emit(h, this |> t)
    case Await(req,recv) => this match {
      case Halt(err) => Halt(err) |> recv(Left(err))  <——如果this停止了，则
      case Emit(h,t) => t |> Try(recv(Right(h)))          进行适当的清理。
      case Await(req0,recv0) => await(req0)(recv0 andThen (_ |> p2))
    }
```

```
    }
}

def pipe[O2](p2: Process1[O,O2]): Process[F,O2] =
    this |> p2
```

我们还会用到帮助函数 kill，它会给一个 Process 最外层的 Await 传递 Kill 异常，但忽略余下的输出。

示例 15.8 kill 帮助函数

```
@annotation.tailrec
final def kill[O2]: Process[F,O2] = this match {
    case Await(req,recv) => recv(Left(Kill)).drain.onHalt {
        case Kill => Halt(End)  <——我们将Kill异常转为正常结束。
        case e => Halt(e)
    }
    case Halt(e) => Halt(e)
    case Emit(h, t) => t.kill
}

final def drain[O2]: Process[F,O2] = this match {
    case Halt(e) => Halt(e)
    case Emit(h, t) => t.drain
    case Await(req,recv) => Await(req, recv andThen (_.drain))
}
```

注意，|> 适用于任何 Process[F, O] 类型，因此它同样适用于 Process1，即 Process[IO,O]，还有后面我们将要讨论的双输入 Process。

有了 |>，我们可以很容易地为 Process 添加函数，以附属各种 Process1 去转换得到输出。比如，filter：

```
def filter(f: O => Boolean): Process[F,O] =
    this |> Process.filter(f)
```

还有 take 和 takeWhile 等。在章节对应的源码里可以看到更多的例子。

15.3.4 多个输入流

想象一下，我们需要将两个全是华氏温度的文件，f1 和 f2 混在（zip）一起，将每个值变成摄氏温度，并做四舍五入，最后一次性输出到文件 celsius.txt。

这类场景可以用到泛化的 Process 类型。与 Process1 类似，具化一个 Process 的新类型 Tee，它可以合并两个输入流。[9] 这次也一样，我们简化了 F：

```
case class T[I,I2]() {
    sealed trait f[X] { def get: Either[I => X, I2 => X] }
    val L = new f[I] { def get = Left(identity) }
    val R = new f[I2] { def get = Right(identity) }
```

9 *Tee* 命令来自字母 T，其外形差不多就是两个输入汇成一个输出的样子。

```
}

def L[I,I2] = T[I,I2]().L
def R[I,I2] = T[I,I2]().R
```

这和之前的 `Is` 类似，差别在于它有两个可能的值，`L` 和 `R`，并用 `Either[I => X, I2 => X]` 在模式匹配的时候用以区分这两种类型。[10] 有了 `T`，我们就可以定义类型别名 `Tee`，用于接收两种类型时的输入：

```
type Tee[I,I2,O] = Process[T[I,I2]#f, O]
```

照例，我们还要定义一下常规函数（convenience function），以便构建这些特定的 `Process` 类型。

示例 15.9 给 Tee 的输入用的常规函数

```
def haltT[I,I2,O]: Tee[I,I2,O] =
    Halt[T[I,I2]#f,O](End)

def awaitL[I,I2,O](
      recv: I => Tee[I,I2,O],
      fallback: => Tee[I,I2,O] = haltT[I,I2,O]): Tee[I,I2,O] =
    await[T[I,I2]#f,I,O](L) {
      case Left(End) => fallback
      case Left(err) => Halt(err)
      case Right(a) => Try(recv(a))
    }

def awaitR[I,I2,O](
      recv: I2 => Tee[I,I2,O],
      fallback: => Tee[I,I2,O] = haltT[I,I2,O]): Tee[I,I2,O] =
    await[T[I,I2]#f,I2,O](R) {
      case Left(End) => fallback
      case Left(err) => Halt(err)
      case Right(a) => Try(recv(a))
    }

def emitT[I,I2,O](h: O, tl: Tee[I,I2,O] = haltT[I,I2,O]): Tee[I,I2,O] =
    emit(h, tl)
```

我们再定义一些 `Tee` 组合子。链状混合（Zipping）是 `Tee` 特有的一种情况，先从左边读取，然后读取右边（或相反的顺序），最后递送这一对值。注意，我们需要明确两种输入读取的顺序，在 `Tee` 与有着外部影响的流产生交互时，这是一种很重要的能力：[11]

```
def zipWith[I,I2,O](f: (I,I2) => O): Tee[I,I2,O] =
    awaitL[I,I2,O](i  =>
```

---

10 `Either` 里的函数 `I => X` 和 `I2 => X` 是一种等价见证（equality witness）形式，即证明一种类型等于另一种类型。

11 我们可能并不希望明确这种顺序，允许驱动器去做出不确定的选择，也允许驱动器能够同时执行这两种选择，更多内容请见章节备注。

```
    awaitR        (i2 => emitT(f(i,i2)))) repeat

def zip[I,I2]: Tee[I,I2,(I,I2)] = zipWith((_,_))
```

这个转换器会在任意一边穷尽时终止，和 List 的 zip 函数一样。其他还有很多 Tee 的组合子是我们可以实现的，比如当读取左边满足某个条件时转去读取右边；又或者，左边读取 5 个值，然后右边读取 10 个；还可以左边读取一个值后，根据这个值来决定读取多少右边的，等等。

将两个过程传递给一个 Tee 应该最需要做的事情，我们可以在 Process 上定义一个函数用 Tee 来合并两个过程，看上去和 |> 很像。并且这个函数适用于任何 Process。

示例 15.10 tee 函数

```
def tee[O2,O3](p2: Process[F,O2])(t: Tee[O,O2,O3]): Process[F,O3] =
    t match {                                    若t停止了，则优雅地干掉两边的输入。
        case Halt(e) => this.kill onComplete p2.kill onComplete Halt(e)  ←┘
        case Emit(h,t) => Emit(h, (this tee p2)(t))  ←递送首部的值，然后递归。
        case Await(side, recv) => side.get match {  ←确认是左边，还是右边的请求。
            case Left(isO) => this match {  ←这是左边的Process，接收O。
                                        Tee是来左边输入的请求，此时是停止的，则做停止处理。
                case Halt(e) => p2.kill onComplete Halt(e)  ←┘
                case Emit(o,ot) => (ot tee p2)(Try(recv(Right(o))))  ←将可用的值
                                                  当前没有任何可用的值，继续  喂给Tee。
                case Await(reqL, recvL) =>  ←等待而后进行tee的操作。
                    await(reqL)(recvL andThen (this2 => this2.tee(p2)(t)))
            }
            case Right(isO2) => p2 match {  ←这是右边的Process，接收O2，
                                              除此之外与上面的一样。
                case Halt(e) => this.kill onComplete Halt(e)
                case Emit(o2,ot) => (this tee ot)(Try(recv(Right(o2))))
                case Await(reqR, recvR) =>
                    await(reqR)(recvR andThen (p3 => this.tee(p3)(t)))
            }
        }
    }
```

15.3.5 去向

如何使用 Process 类型进行输出呢？我们将经常想要把 Process[IO,O] 的输出发送到某个地方（比如说文件）。也许你会意外，我们可以将去向（sink）看作递送函数的过程：

```
type Sink[F[_],O] = Process[F[_], O => Process[F,Unit]]
```

这样做是合理的，Sink[F[_],O] 提供了一系列函数对输入类型 O 进行处理，这些函数返回 Process[F, Unit]。看一个将字符串写入文件的 Sink：

```
def fileW(file: String, append: Boolean = false): Sink[IO,String] =
    resource[FileWriter, String => Process[IO,Unit]]
        { IO { new FileWriter(file, append) }}
        { w => constant { (s: String) => eval[IO,Unit](IO(w.write(s))) }}
```

```
    { w => eval_(IO(w.close)) }

def constant[A](a: A): Process[IO,A] = ←——无限的常量流。
  eval[IO,A](IO(a)).repeat
```

如你所见，这看起来很简单，但你有没有注意到这里缺少了什么。这里没有了异常处理代码，没有了组合子在异常发生或 Sink 的供给结束时，保证 FileWriter 会被关闭。

我们可以用 tee 实现组合子 to，可以用来将输出传给 Sink：

```
def to[O2](sink:: Sink[F, O]): Process[F, Unit] =
  join { (this zipWith sink)((o,f) => f(o)) }
```

◈ **练习 15.12**

to 中使用了一个新组合子 join，让 Process 连接一个内嵌的 Process。请用存在基元函数实现 join，你会发现它在之前章节中貌似出现过。

```
def join[F[_],O](p: Process[F, Process[F,O]]): Process[F,O]
```

有了 to，我们的程序可以写成这样：

```
val converter: Process[IO,Unit] =
  lines("fahrenheit.txt").
  filter(!_.startsWith("#")).
  map(line => fahrenheitToCelsius(line.toDouble).toString).
  pipe(intersperse("\n")).
  to(fileW("celsius.txt")).
  drain
```

上面用到的帮助函数 drain，目的是忽略 Process 的所有输出：

```
final def drain[O2]: Process[F,O2] = this match {
  case Halt(e) => Halt(e)
  case Emit(h, t) => t.drain
  case Await(req,recv) => Await(req, recv andThen (_.drain))
}
```

一旦通过 runLog 执行，converter 便会打开输入文件和输出文件，增量地转换所有输入流，并跳过注释的行。

15.3.6 Effectful 通道

我们可以泛化 to，允许其结果的类型不只是 Unit。其实现其实是一样的！你会发现就是操作的类型比之前更宽泛。我们就把这种更宽泛的操作叫作 through：

```
def through[O2](p2: Process[F, O => Process[F,O2]]): Process[F,O2] =
  join { (this zipWith p2)((o,f) => f(o)) }
```

对于这种模式，我们引入一个类型别名：

```
type Channel[F[_],I,O] = Process[F, I => Process[F,O]]
```

当处理管道需要执行一些 I/O 行为时，Channel 显得尤为有用。一个典型的例子就是，一个应用需要执行数据库查询，查询的结果最好是 Process[IO, Row]，其中 Row 表示数据库的一行记录。这样一来，所有的程序就可以对查询的结果集用上所有华丽的流式转换器。

下面是一个简单查询执行器的签名，其中用 Map[Srtring,Any] 来表示查询结果的类型（更多内容参见章节源码的实现）：

```
import java.sql.{Connection, PreparedStatement, ResultSet}

def query(conn: IO[Connection]):
    Channel[IO, Connection => PreparedStatement, Map[String,Any]]
```

我们完全可以写成 Channel[PreparedStatement, Source[Map[String, Any]]]，可为什么不这么做呢？因为我们不希望使用 Channel 的代码关心如何获取 Connection（用于创建 PreparedStatement）。一旦由 Channel 管理这种依赖，便可以保证在执行完查询后关闭其连接。

15.3.7 动态资源分配

现实编程中，当需要转换某些输入流时，我们需要动态地分配资源。比如，我们可能遇到下面的场景：

- *动态资源分配*——读取文件 fahernheits.txt，将其中所有的文件名对应的文件连接成一个逻辑的流，并将这个流的数据转换成摄氏温度，最后写入文件 celsius.txt。
- *多去向输出*——很像动态资源分派，但不是创建一个输出文件，而是根据输入文件 fahernheits.txt 中每个文件名追加 .celsius，并创建对应的输出文件。

这样的能力可否合并到 Process 的定义中还能保证资源安全呢？答案是，能！事实上我们已经具备这样的能力，使用 flatMap 组合子便能做到。

比方说，flatMap 加上其他已经存在的组合子就可以帮我们解决第一个场景：

```
val convertAll: Process[IO,Unit] = (for {
    out <- fileW("celsius.txt").once ←——至多只取一个输出的流；细节请看章节代码。
    file <- lines("fahrenheits.txt")
    _ <- lines(file).
         map(line => fahrenheitToCelsius(line.toDouble)).
         flatMap(celsius => out(celsius.toString)).
} yield ()) drain
```

上面代码完全是资源安全的，所有文件句柄都在结束之后自动关闭，哪怕是出现了异常。遇到的任何异常也将在 runLog 函数调用时抛出来。

多文件输出的时候，我们只需要改变一下调用 flatMap 的顺序即可：

```
val convertMultisink: Process[IO,Unit] = (for {
    file <- lines("fahrenheits.txt")
    _ <- lines(file).
         map(line => fahrenheitToCelsius(line.toDouble)).
         map(_ toString).
         to(fileW(file + ".celsius"))
} yield ()) drain
```

当然，我们还可以在任何位置附加各种转换、映射、过滤等：

```
val convertMultisink2: Process[IO,Unit] = (for {
    file <- lines("fahrenheits.txt")
    _ <- lines(file).
         filter(!_.startsWith("#")).
         map(line => fahrenheitToCelsius(line.toDouble)).
         filter(_ > 0). // ignore below zero temperatures
         map(_ toString).
         to(fileW(file + ".celsius"))
} yield ()) drain
```

更多代码示例可以在章节源码中找到。

15.4 应用场景

本章呈现的想法有着广泛的应用场景。有相当惊人数量的程序都可以被转换成流式处理，一旦你意识到了这种抽象，你会发现到处都是。下面就是一些典型的应用场景：

- 文件 I/O——我们已经演示过如何对文件 I/O 进行流式处理。尽管示例集中在逐行的读写上，但是我们的库也同样可以处理二进制文件。
- 消息流、状态机、actor——大型系统通常通过消息传递的通信方式来对系统组件进行解耦。系统就像 *actor* 一样，通过消息的收发进行通信。我们可以将这些组件视作流式处理器来架构。这样使得我们能够描述机器复杂的状态机，用高级别的可组合的 API 实现其行为。
- 服务器、web 应用——一个 web 应用可以被理解为将 HTTP 请求流变成 HTTP 响应流的转换器。
- UI 编程——我们把 UI 事件如鼠标点击视作事件流，而 UI 就是一个流式处理器，以对用户的交互做出响应。
- 大数据、分布式系统——流式处理的库可以被分布和并行，用以处理巨大的数据。关键点在于这些流式处理的网络节点是没必要在同一台机器上的。

若你想要学习更多的应用，查看章节备注可以获取额外的讨论和更深入的内容。章节备注和源码同样讨论了一些对 `Process` 类型的扩展，包括非确定性选择以允许并发地对 `Process` 执行进行求值。

15.5 小结

我们是以一个前提开始本书的，就是假定程序仅使用纯函数。伴随着这个唯一的前提和影响，我们完整地开发了一种全新的编程方法。在最后的章节，我们还构建流式处理和增量 I/O 的库，证明我们能够保持用组合式风格实现书中所有的程序，即便是那些与外界交互的程序。其中无论大小，我们都使用 FP 来实现并完成了。

FP 是个很深的话题，我们仅触及皮毛。现在你已经具备独自继续这个旅程的能力，在工作中更多地实践函数式编程吧。尽管好的设计总是困难的，随着时间推移你的函数式表达能力会越来越强。应用 FP 解决问题一旦多起来，你还将发现新的模式和更强大的抽象。

享受这旅程吧，保持学习，祝你好运！

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Scala函数式编程

函数式编程是这样一种软件开发风格：它强调函数不依赖于程序状态。用函数式开发，代码更容易测试和复用，更容易并行化，且比其他代码更不容易产生错误。

Scala是一门新兴的JVM语言，对函数式编程提供了强有力的支持。与Java相近的语法，以及透明的互操作性使得Scala成为一门很适合学习的函数式编程语言。

《Scala函数式编程》对渴望学习函数式编程并将其应用在实际工作中的程序员来说，是一本重要的教程。这本书用合理、清晰的步骤将读者从基础知识引导到高阶话题。在里面你会发现一些具体的例子和练习，它们将帮你开启通向函数式编程世界的一扇门。

本书内容包括：

- 函数式编程概念
- 为什么要用函数式编程以及如何用函数式编程
- 如何写面向多核的程序
- 练习并检验你的理解

这本书假定读者之前没有函数式编程的经验。如果之前有Scala或Java的经验会对理解更有帮助。

“让你洞察计算的本质。”
—— Martin Odersky, Scala的作者

“Scala和Java8开发者的函数式编程权威指南！”
—— William E. Wheeler, TekSystems

“本书向你展示了提升Scala技能的方法和理念，它已超越‘更好的Java’。”
—— Fernando Dobladez, Code54

“里面的练习有些挑战，很有趣，对你在真实世界中使用它很有益。”
—— Chris Nauroth, Hortonworks

“边干边学，而非只是阅读。”
—— Douglas Alan、Eli和Edythe L.Broad，哈佛和麻省理工学院

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

MANNING

责任编辑：张春雨
封面设计：吴海燕

上架建议：函数式编程

ISBN 978-7-121-28330-7
9 787121 283307 >
定价：69.00元